Emulating C++0x Concepts

Andrew Sutton[a], Jonathan I. Maletic[a]

[a] *Kent State University*
Department of Computer Science
Kent, Ohio 44242

Abstract

A library for the emulation of C++0x concepts developed using the emerging C++11 programming language is presented. The library integrates existing techniques for concept checking and template metaprogramming to provide a uniform interface to defining and using concepts. The purpose of this work is to establishing a concrete foundation for experimentation of design techniques for concepts and to motivate and evaluate language design. The viability of the approach is demonstrated by applying it to characterize a number of previously identified usability problems with concepts in the proposed C++0x language. In particular, issues related to the use of explicit and automatic concepts in generic library design from the perspective of this experiment are examined. Issues related to concept refinement, default implementations of requirements, and the generation of error messages are also discussed.

Keywords: C++, Concepts, Generic Libraries

1. Introduction

Concepts describe the abstractions operated on by generic data structures and algorithms in C++. They are at the heart of every generic library, often as accompanying documentation, sometimes as statically checked constraints on template arguments, and implicitly when not otherwise stated. Over the past decade, a great deal of effort has been exerted to codify concepts into the C++ programming language. This standardization effort was ultimately unsuccessful (Stroustrup, 2009a). Despite the fact that concepts are not a part of the C++11 programming language, this does not imply that concepts will not continue to play a vital role in the definition, development, and maintenance of C++ generic libraries. In fact, concepts may return to the C++ language in as little as five years (Stroustrup, 2009a).

A principle reason for removal of concepts from C++0x is the lack of practical feedback from programmers. Although one compiler supporting a subset

Email addresses: asutton@cs.kent.edu (Andrew Sutton), jmaletic@kent.edu (Jonathan I. Maletic)

of the proposed language extensions was produced (Gregor, 2009), the project is now unmaintained and suitable only for small-scale experimentation; it is not viable for large or complex software projects. The lack of industry-scale solutions for working with concepts makes widespread adoption virtually impossible. The absence of widespread developer adoption has led to an absence of practical feedback in the design and implementation of concept systems as constraints for template arguments. Such feedback would, in turn, better motivate the requirements for a language design.

The goal of our work is to support and foster experiments in generic library design and, in particular, concept design. To this end, we developed a library-based approach for experimenting with the design and application of concepts in generic libraries by emulating several features of the proposed C++0x extensions. Our approach does not mandate any specific style with regards to the definition and application of concepts. Instead, we aim to support a range of styles to better facilitate experimentation with varied design techniques. Ultimately, we hope to leverage such experiments to help motivate and empirically evaluate future language designs for concepts.

We opted for the library-based approach over building a preprocessor or directly designing language features because those require some commitment to syntax and semantics. Our primary motivation is understanding and documenting the trade-offs that generally accompany language design.

The library produced as a result of this work, the Origin Concept library, integrates a number of existing techniques for generic programming and template metaprogramming in order to provide support for defining concepts and concept maps, implicit and explicit template argument checking, concept-based overloading, the provision of default operations, and the definition of axioms. The implementation is supported by many of the new language and library features in C++11. However, the library is not intended to be a "drop-in" replacement for true language support. The proposed extensions for C++0x concepts were complex and addressed a number of issues that we see as being outside the scope of our work. For example, C++0x concepts require separate type checking for template definitions (separate from their instantiations). Although the Origin library does not directly address this requirement, we do discuss a known (partial) solution in Section 4.4.3. C++0x concepts also included changes to lookup rules in constrained templates, which addressed some issues with argument dependent lookup (ADL). We did not try to address this since there are alternative approaches to dealing with ADL problems. Our primary goal focuses on issues related to template argument checking.

We validate applicability of the library by applying it to reproduce a number of design questions that arose during the development of the C++0x concepts proposals, especially those that we feel will have a substantial impact on the way programmers use concepts. This validation approach is not meant to demonstrate viable solutions to these questions, but rather to show that our emulation approach is capable of representing a broad spectrum of solutions to such problems. In essence, we hope to use the emulation approach to better explore the design space of concepts, both as library components and as a language feature.

The primary contribution of this work is the creation of infrastructure to support experiments on the design, implementation, and evaluation of systems of concepts for generic libraries. This infrastructure builds on and integrates existing techniques for generic programming, concept emulation, and template metaprogramming to provide a set of features that support these goals. This project extends previous work on concept emulation by incorporating more recent results of the corresponding language design effort and translating them into a new setting, C++11. New features in the C++11 programming language and Standard Library provide a more effective platform for conducting these investigations.

In Section 2, we present work related to the design, specification, and emulation of concepts for C++. Section 3 gives an overview of proposed features for C++0x concepts, and Section 4 presents the techniques we use to emulate many of the proposed features. In Section 5, a brief overview of the Origin Libraries, the experimental infrastructure used to support this and future projects is given. We discuss issues related to the design and use of concepts in Section 6, and present our conclusions in Section 7.

2. Related Work

The specification of concepts in their current form can be attributed to the Standard Template Library (STL) and its development (Musser and Stepanov, 1994). Concepts for these libraries are specified as part of the documentation that classifies the elements of the library according to their syntax and semantics (Austern, 1998). These concepts are used to express requirements on template arguments of generic algorithms and data structures and are expressed as sets of associated types, valid expressions, semantics, invariants, and performance guarantees.

The broader adoption of the generic programming style inspired several methods for implementing constraints on template parameters. Stroustrup reports on early experiments to constrain template parameters through inheritance based approaches and use patterns (Stroustrup, 1994). Siek and Lumsdaine apply this technique to define constraint classes for the concepts required by the STL [6] (Siek and Lumsdaine, 2000). A technique described by Zólyomi and Porkoláb relies on advanced C++ template programming techniques to emulate concept checks as template metafunctions and static assertions (Zólyomi and Porkoláb, 2004). Our approach to concept emulation unifies aspects of both the use-pattern and metaprogramming approaches to provide a number of features in this space.

McNamara and Smaragdakis develop a technique for emulating the specification and requirement of static interfaces (McNamara and Smaragdakis, 2000). The notion of static interface is considerably more restrictive than the use-pattern based constraints and is used to support strict checks on interface conformance in contrast to the less strict approach of checking the availability of valid expressions and associated types. Also, the method is intrusive; it necessitates changes to the design and implementation of generic data structures and

algorithms. Moreover, the goal of the static interfaces work was to enforce strict conformance to concepts. Our perspective is that constraints should be written loosely to accommodate a wide range of valid implementations. Strict type conformance with static interfaces prevents, e.g., the use of expression templates (Veldhuizen, 1995) in generic code.

While the library-based approaches provide useful capabilities for constraining template arguments, they also demonstrate a fundamental need for language support. Along these lines, comparative studies of languages supporting generic programming also helped elicit requirements (Garcia et al., 2003; Siek and Lumsdaine, 2005). In 2005, two competing proposals for integrating concepts into the C++ language were introduced: one from Indiana University (Siek et al., 2005) and the other from Texas A&M University (Stroustrup and Dos Reis, 2005; Dos Reis and Stroustrup, 2006). The C++0x proposal for concepts evolved from the ideas put forth in these different approaches (Gregor et al., 2006) and resulted in a prototype compiler, ConceptGCC (Gregor, 2009). In 2009, the C++ committee voted to remove the proposal from future versions of the Draft Standard because there were a number of unanswered technical and usability questions that may have hindered adoption of the new language (Stroustrup, 2009a,b).

Although C++0x concepts did not become part of the core language, the premises of the proposed work remain fundamental to the development of generic libraries. David and Haveraaen describe a program transformation technique to transform the C++ concept syntax into source code that can be used to constrain generic algorithms (David and Haveraaen, 2009). This approach translates the concepts and concept maps from the proposed syntax into a series of template specializations with the intent of emulating the changes to the name lookup rules imposed by C++0x concepts. When written by hand, the implementation is highly intrusive; it requires significant changes to existing template definitions. A program transformation can hide the added complexity.

Our choice to use emulation is largely motivated by practical concerns. The cost of modifying an existing compiler or building a preprocessor is nontrivial, especially for a language as large and complex as C++. Furthermore, such approaches require us to make concrete decisions about the syntax and semantics of the language before implementing a solution and conducting experiments. In contrast, the emulation approach allows us to work directly in the "mechanics" of the emulated features. We have a greater degree of freedom to experiment with and evaluate different designs before making final decisions about the semantics of the language. The obvious drawback to the approach is that we cannot emulate features that require language support (e.g., modifying lookup rules). However, we feel that the approach is sufficient for studying a broad range of design problems in the context of concepts.

3. Programming with Concepts

In this section, we give an overview of the syntax and semantics of C++0x concepts. This is not a complete overview of the proposal. We omit the descrip-

tion of features related to the proposed lookup rules because they do not play a substantial role in this work (they are not easily emulated without additional effort on the part of end programmers or a preprocessor).

3.1. Concepts and Concept Maps

A concept defines a set of requirements on template arguments. A concept requirement is written, for example, as Iterator<T>. Here, Iterator is a concept name, and T is a template argument. The requirement is valid if T satisfies all of Iterator's requirements. If any of the requirements are not met (e.g., a type name cannot be deduced, an overload is not found, or a nested requirement is invalid), then the model is invalid. We say that T models Iterator if Iterator<T> is valid, or that T is a model of Iterator, or more briefly that T is an Iterator.

The Iterator concept, as defined by the C++0x Draft Standard (Becker, 2009), is written:

```
concept Iterator<typename X> : Semiregular<X> {
  typename reference = X::reference;
  typename postincrement_result;

  reference operator*(X&);
  X& operator++();
  postincrement_result operator++(int);

  requires MoveConstructible<reference>;
  requires MoveConstructible<postincrement_result> &&
           HasDerefeference<postincrement_result>;
}
```

This concept defines two associated types, reference and postincrement_result, which denote the result types of those operations. Here, reference is assigned a default type while postincrement_result is deduced as the result of the post-increment operator. The concept also declares associated function requirements, operator* and operator++. The concept also contains a number of nested associated requirements on the previously declared associated type names. Here, reference and postincrement_result are both required to satisfy the requirements of the MoveConstructible concept, and postincrement_result is further required to model HasDereference.

Finally, the Iterator concept is refined from another concept, SemiRegular. Iterator includes all of the requirements of the SemiRegular concepts. Refinement is similar to class inheritance in that the refining concept includes the associated type names and operations from the refined concept. Refinement also defines a partial ordering on concepts similar to the subtype relation defined by inheritance. Models of concepts are also ordered by this relation, which is used to support concept-based overloading.

A concept check of the form, Iterator<T>, implicitly generates a concept map if Tsatisfies the requirements of Iterator. A concept map is a binding between the concept, its arguments, storing the associated types and operations that satisfy concept's requirements.

A concept map can also be explicitly defined to adapt a type to a concept specification. For example, pointers are valid models of random access iterators, but fail to meet the requirements of the RandomAccessIterator concept definition because some associated types cannot be automatically deduced (e.g., value_type). As such, we can define a concept map that explicitly binds pointer types (T*) to the RandomAccessIterator concept, providing the appropriate names for the required associate types:

```
template<typename T>
concept_map RandomAccessIterator<T*> {
  typedef T value_type;
  typedef T& reference;
  // ...
};
```

Using this definition, a concept check of the form RandomAccessIterator<int*> will refer to this concept map rather than an implicitly generated one (which would not succeed). Note that since RandomAccessIterator is a refinement of other iterator concepts, this concept map will also satisfy the requirements of the refined concepts. That is, if RandomAccessIterator<int*> is a valid model, then so is Iterator<int*>.

3.2. Associated Semantics

The semantics of a concept are defined in two ways: a default implementation of a required operation, or a set of axioms constraining the behavior of a required operation. For example, the EqualityComparable concept is defined as:

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

  axiom Reflexive(T a) { a == a; }
  axiom Symmetric(T a, T b) { (a == b) => (b == a); }
  axiom Transitive(T a, T b, T c) { (a == b && b == c) => (a == c); }
}
```

The != operator is a required operation with a default definition written in terms of ==. The concept assigns a canonical meaning to !=, provided that a definition of == is available.

The meaning of equality is defined by the axioms Reflexive, Symmetric, and Transitive. These state the required semantics of ==, namely that it is an equivalence relation. Here the operator, => denotes logical implication. If the expression on the left is true, then the condition on the right must also be true.

3.3. Constrained Templates

Concepts constrain template arguments by limiting the set of types that can be used to instantiate the template. Consider the find algorithm below.

```
template<InputIterator Iter, typename T>
   requires InputIterator<Iter> &&
            EqualityComparable<typename InputIterator<Iter>::value_type, T>
Iter find(Iter first, Iter last, T const& x) {
   while (first != last && *first != x)
     ++first;
   return first;
}
```

The requires clause lists a conjunction of concept requirements each of which must be validated before the compiler instantiates the template. This is called *concept checking*. If the concept check fails, then a compiler error is emitted at the point of failure. Concept checking gives the compiler the ability to emit more meaningful and concise error messages for generic code. For example, suppose a programmer unintentionally causes Iter to be substituted with int. An int is not an InputIterator because it cannot be dereferenced (i.e., it has is no unary operator *). Since the error is caught by concept checking, the resulting message may read, "InputIterator<int> is not a valid model because..." This is far better than the current situation where such errors might be reported as, "no operator* in the expression *first == x."

Class templates and member functions can also be constrained using concepts. A partial listing of the pair template and its requirements is given as:

```
template<VariableType T, VariableType U>
class pair {
   requires LessThanComparable<T> && LessThanComparable<U>
   bool operator<(pair const& x) const;
};
```

Here, the template parameters T and U are constrained (using a shorthand notation) by the VariableType concept. This is equivalent to writing requires VariableType<T> && VariableType<U>. The member operator < further constrains T and U to be LessThanComparable. Here, the constraints are applied to the member and not the class because the instantiation (and checking) of the operator is dependent upon its use in a program. If the member is not used, then it must not be instantiated. Since the member is conditionally instantiated, it is not appropriate to impose those requirements on the template parameters of the class where they would be checked for any instantiation of pair.

Concepts not only constrain template arguments but also constrain the requiring template definition. This is to say that a template definition should not use operations that are not expressed as a requirement on its template parameters. To do so would open the door for uncaught type errors. For example, suppose we write the accumulate algorithm as:

```
template<InputIterator Iter, typename T>
Iter find(Iter first, Iter last, T init) {
   for( ; first != last; ++first)
      init += *first;
   return init;
}
```

In this algorithm, there are no requirements for the operator +=. Although the compiler can check the iterator parameters, it cannot check the conceptual validity of the expression init += *first. Any errors could only be caught when the instantiated algorithm was type checked. This was not permitted with C++0x concepts. Instead, the compiler was required to check whether or not the algorithm referred to functions outside the set admitted by its requirements.

The check is done by synthesizing *archetypes* from the required concepts. An archetype is essentially a class whose definition is generated by the set of stated requirements of a concept. If an expression in the template is not mapped to an operation in a synthesized archetype, then the template definition contains an error.

3.4. Concept Overloading

Concept overloading allows the compiler to select between overloads of a function based on the concepts modeled by the template arguments. Consider the following overloads:

```
template<InputIterator Iter>
int distance(Iter first, Iter last) {
   int n = 0;
   for( ; first != last; ++first)
      ++n;
   return n;
}

template<RandomAccessIterator Iter>
int distance(Iter first, Iter last) {
   return last - first;
}
```

The two overloads of the distance function are distinguished only by the concepts required of the Iter template parameters. When a call to the distance function is found, the compiler must instantiate both overloads, which involves checking their concept requirements. There are three possibilities:

1. The template arguments satisfy neither of the requirements, so no best overload can be determined. A compiler error is emitted.
2. The template argument is an InputIterator but not a RandomAccessIterator. The first overload is selected.
3. The template argument is a RandomAccessIterator (and implicitly an InputIterator). The compiler chooses the overload based on the most specialized (refined) requirements, in which case the second overload is chosen.

In other words, the partial ordering of function overloads is extended for concepts as well as types. This is determined by the refinement relation.

3.5. Concept Checking

Concept checking is predicated on the ability of the compiler to find a concept map that binds a concept to its template arguments. If a concept map can be found, the check is successful; if not, the check fails. There are two approaches to the lookup of concept maps: explicit and automatic.

An *explicit concept* is one that requires the programmer to explicitly provide a concept map in order to satisfy the requirements of that concept. Intuitively, this is not much different than listing a set of implemented interfaces for a class (e.g., in Java). For legacy code, however, explicit concepts require a maintainer to retroactively specify concept maps for any usage of generic libraries or risk breaking existing code. The burden of explicit concept checking by data structure designers and end users may be substantial, especially if a data structure models a large number of concepts.

An *automatic concept* is one whose requirements are evaluated on provided template arguments by the compiler, implicitly generating a concept map if all requirements are satisfied. Although this approach may seem like a more feasible solution, it introduces ambiguities; a type may define operations required by a concept without being a model of the concept. For example a stopwatch class may define operations begin and end, which would have little in common with the begin and end methods defined by STL container classes (e.g., list, vector, etc.).

Moreover, many concepts have semantic properties that are not statically checked by C++ compilers. This issue is readily illustrated by two overloads of the vector class' range constructor. Partial definitions of the InputIterator and ForwardIterator concepts and the two overloads are:

```
concept InputIterator<typename X> {
   // requires basic iterator functionality
}

concept ForwardIterator<typename X> : InputIterator<X> {
   axiom MultiPass(X a, X b) {
      a == b => ++a == ++b;
   }
}

template<typename T>
class vector {
public:
   template<InputIterator Iter>
   vector<T>::vector(Iter f, Iter l) {
      for( ; f != l; ++f) push_back(*f);
   }

   template<ForwardIterator Iter>
   vector<T>::vector(Iter f, Iter l) {
      resize(distance(f, l));
      copy(f, l, begin());
```

```
  }
}
```

With automatic concepts, the compiler is required to automatically evaluate the difference between the InputIterator and ForwardIterator. Here, this is impossible because ForwardIterator and InputIterator share the same syntactic requirements. The only differentiating feature is a semantic requirement, the MultiPass axiom, and that cannot be statically evaluated. With only automatic concept checking, this example could easily result in runtime errors. We must explicitly differentiate concepts that are syntactically equivalent using explicit concepts.

What appears to be needed is a combination of implicit and explicit concepts. By default, concepts can be automatically checked, putting as little burden as possible on type providers. In cases where concepts cannot be syntactically differentiated, explicit concepts must be used. For illustration, Stroustrup suggests using explicit refinement to define ForwardIterator (Stroustrup, 2009b).

```
concept ForwardIterator<typename X> : explicit InputIterator<X> {
  axiom MultiPass(X a, X b) ...
}
```

The explicit qualifier prevents the compiler from considering X as a ForwardIterator unless a concept map specifically enabled the conversion. This allows the writing of concept systems that are largely based on automatic concepts, and explicit differentiation can be used when needed.

4. Emulating Concepts

In this section, we present our integrated approach to concept emulation for the Origin libraries. The aim of the library is to support library writers in

- defining automatically and explicitly checked concepts,
- defining concept maps,
- providing default implementations of required operations,
- writing axioms,
- constraining template arguments, and
- supporting concept-based overloading and specialization.

By using existing language features, we can discover the limits of what can be done without changing the semantics of the language. We can, with some limitations, reason about software design techniques and help produce rationale for future language extensions. The obvious downsides of the emulation approach are the lack of more elegant syntax and the absence of integration with the compiler (e.g., for error reporting).

There are two alternative approaches. First, we could have chosen to write a preprocessor, however, this is more complex than it seems. The preprocessor would have to understand C++ (a non-trivial proposition to begin with), we would have to commit to a syntax, and the preprocessor would have to generate code that actually implemented the concept checking and overload support already provided by the Origin library. In other words, our library is a possible first step towards building a viable preprocessor. Second, we could have modified the language directly. As with the preprocessor approach, this would require us to commit to a syntax and semantics. One of our goals is to provide support for designing language features related to concepts. We are not yet trying to design language features; we are trying to build a framework to help explore trade-offs.

There are aspects of the proposed C++0x concepts features whose emulation is not addressed by the Origin Concept library. We specifically omit support for features that modify the C++ name lookup rules and those concerned with template definition checking (i.e., archetypes). The focus of our approach targets the specification and application of requirements to template arguments. While these omitted features may be required for a more complete definition of concepts as language feature, we do not believe that they are essential for conducting the kinds of experiments in which we are currently interested.

4.1. Foundations

Our approach to emulating concepts is largely based on the integration of two different techniques found in current generic libraries: constraint classes and type traits. Here, we give an overview of the techniques and discuss how we integrate and extend those ideas to support concept emulation.

4.1.1. Constraint Classes

The notion of using class templates to constrain template arguments is not new. The basic techniques have been employed essentially since templates were first added to C++ (Stroustrup, 1994). These techniques are also the basis of the Boost Concept Check Library (Siek and Lumsdaine, 2000), which is used in a number of libraries in the Boost C++ Libraries and also in GCC's libstdc++ (Standard Library implementation).

The premise of the technique is to check a concept's specified valid expressions by instantiating them in a constrained template. A concept is represented by a class template that has a member function containing the valid expressions. The class template is instantiated in order to force the compiler to resolve lookups in the nested valid expressions. If any type errors are caught within the constraints class, compilation fails and an error message is emitted. We note that the use of valid expressions within constraint classes to check template arguments is, in many respects, analogous of the use-pattern approach to specifying concepts proposed by Dos Reis and Stroustrup (Dos Reis and Stroustrup, 2006).

Our approach to writing constraints classes is structurally different than that used in the Boost Concept Check Library, although largely equivalent. Consider a constraint class that checks for the existence of the == operator:

```
template<typename T, typename U>
struct HasEqual<T, U> {
   HasEqual() { auto p = constraints; }

   static void constraints(T x, U y) {
      x == y;
   }
};
```

The HasEqual constructor forces the instantiation of the static constraints member function by declaring a pointer to the function. This function implements a set of valid expressions for the objects introduced by its formal parameters. The body of this constraints function contains a single statement that forces a lookup on an appropriate == for types of T and U. If no such operator can be found, instantiation will fail, and the compiler emits the appropriate error message. The result of the expression is unconstrained in this example.

Constraint classes can be reused through either inheritance or composition. For example, consider a definition of the EqualityComparable concept:

```
template<typename T>
struct EqualityComparable : HasEqual<T, T> {
   EqualityComparable() { auto p = constraints; }

   static void constraints(T x, T y) {
      Convertible<decltype(x == y), bool>{};
      Convertible<decltype(x != y), bool>{};
   }
};
```

The inheritance of HasEqual causes the constraint to be checked before the constraints in the EqualityComparable constraint class by virtue of the initialization order. The EqualityComparable class applies additional constraints within its body. Here, the Convertible constraint is written as an explicit initialization of a temporary object. This checks if the result of an expression (the decltype operator returns the type of an expression) can be converted to the specified type, here, bool. A similar requirement is made for !=. Convertible is defined as as:

```
template<typename T, typename U>
struct Convertible {
   Convertible() {
      static_assert(is_convertible<T, U>::value, "Not convertible");
   }
};
```

The constraint is implemented by statically asserting the is_convertible type trait. A type trait is a class template that evaluates some property of its type

arguments. The nested value contains the result of the evaluation. A constraints function is not needed since the trait can be evaluated without introducing objects.

Using constraint-checking classes to constrain template parameters is virtually identical to their use as constraints. In function templates, constraint classes are written as initialized temporary objects (as in the EqualityComparable constraints function), and in class templates they are written as base classes (as in the base class specifiers of EqualityComparable). This is demonstrated by the min algorithm and less function object in Table 1.

Table 1: Constraint classes used in (a) function templates and (b) class templates.

(a)

```
template<typename T>
T const& min(T const& x, T const& y) {
    LessThanComparable<T>{};
    return x < y ? x : y;
}
```

(b)

```
template<typename T>
struct less : LessThanComparable<T> {
    bool operator()(T const& x, T const& y) const {
        return x < y;
    }
};
```

In contrast to our approach, the Boost Concept Check Library, objects are introduced as member variables and the constraints are written in the class's destructor. The behavior is virtually identical, except perhaps in the order in which constraints are checked. We also note that the Boost Concept Check Library should not be used as a base-class constraint (as shown in Table 1). Because objects are introduced as member variables, this can (dramatically) affect the size of constrained classes, leading to larger runtime objects.

4.1.2. Type Traits

Type traits are one of the fundamental building blocks of modern generic libraries. They are class templates that are used to evaluate properties of their type arguments and often to derive or associate types with those arguments. The is_convertible template used in the previous section is an example of a type trait. Another example, is_same is defined below:

```
template<typename T, typename U>
struct is_same { static constexpr bool value = false; };

template<typename T>
struct is_same<T, T> { static constexpr bool value = true; };
```

The type trait is evaluated by writing is_same<X, Y>::value, and the result of that expression is only true when X and Y name the same type. The primary

template (the first declaration) defines the result when X and Y are different types. The specialization defines the result when the template arguments are the same. The resulting value is a constant expression that can be evaluated by the compiler in contexts such as a static_assert.

While many such type traits can be implemented in this fashion (with varying degrees of complexity), a small number require intrinsic compiler support. For example, determining if a class is polymorphic requires a level of introspection that can only be implemented with compiler support.

There is also a category of type traits that can be used to query properties of arbitrary expressions: the result of the expression and whether or not it is. For example, the equal_result type trait "safely" deduces the result type of the expression x == y.

```
template<typename T, typename U>
struct equal_result {
private:
   template<typename X, typename Y>
   static auto check(X x, Y y) -> decltype(x == y);

   static substitution_failure check(...);
public:
    typedef decltype(check(declval<T>(), declval<U>())) type;
};
```

The type trait is implemented using an advanced SFINAE (Substitution Failure Is Not An Error) idiom. It is an adaptation of the technique first described by Alexandrescu for writing similar queries (Alexandrescu, 2000). The result type is constructed using decltype by calling the nested static member, check, over an object of type T. The declval function is used to create "fake" objects of an explicitly specified type in the decltype argument.

Writing equal_result<A, B>::type deduces the result of the expression a == b for arguments of type A and B respectively. If there is an overload of == for A and B, then overload resolution requires that the compiler select the first overload of check. If not, the first overload is not a viable candidate (x == y results in a substitution failure), so the compiler chooses the second overload.

We say that the result is "safely" deduced since lookup failures do not result in compiler errors. The substitution_failure acts as a placeholder. This is particularly important since just writing a "naked" decltype to deduce the result of an expression may result in compiler errors when none are desired.

Having safely deduced the result type, determining whether the expression is valid is straightforward:

```
template<typename T, typename U>
struct has_equal {
  static constexpr bool value = !is_same<
    typename equal_result<T, U>::type, substituion_failuire
  >::value;
};
```

The expression has_equal<X, Y>::value is true only when the type trait equal_result does not result in substitution_failure.

The evaluation of this lookup is equivalent to the use of use-patterns in constraint classes. Both the HasEqual constraint class and the has_equal type trait mandate lookups on the same expression: x == y. The difference between the two is the behavior of the compiler when that lookup fails. For constraints classes, an error is emitted. For type traits, the overload is marked non-viable, but compilation continues.

Because type traits are used as compile time functions on types, we require that traits never generate compiler errors. Note that this is the opposite of our expectations for constraint checking classes. The purpose of those classes is to force compiler errors. Also, the equal_result and has_equal traits can be used to evaluate, at compile time, the valid expressions and constraints on those expressions found within the EqualityComparable constraint class. Our approach to concept emulation encompasses both the use of constraint classes to generate more meaningful compiler errors and the extensive use of type traits to facilitate compile-time queries about concepts and their models. In the following sections, we discuss how these ideas are combined to help define and use concepts and constraints.

4.2. Defining Concepts

Our emulation of concepts integrates constraint classes and type traits to address two problems: generating concept-specific compiler errors and supporting concept-based overloading.

In C++0x, the expression SameType<T, U> denotes a requirement where SameType is a concept name, and the T and U are its concept arguments. In the Origin Concept library, SameType is a *concept class* (a class template) with T and U its template arguments. A concept class supports both constraints checking (to generate compiler errors) and type checking (to support concept-based overloading and other metaprogramming applications). The HasEqual concept class (discussed in the previous section) and its corresponding concept definition from C++0x are shown in Table 2.

The two distinct components of the concept class are clearly present. The default constructor and static constraints functions implement a constraint class exactly as described in the previous section. This aspect of the concept class is used to force compiler errors if the concept's requirements are not satisfied.

The requirements, type, and value members are used to support concept overloading. The requirements tuple is a type list (Alexandrescu, 2001) of type traits and other concepts required by the concept class. Here, has_equal and equal_result are exactly the type traits described in the previous section. The SameType concept is also a requirement of the concept. The has_equal and SameType requirements are evaluated by the concept_check metafunction as a conjunction of type traits. The SameType concept may be implemented using a similar design, or it may be an alias to the is_same type trait.

We note that the requirements tuple is not strictly necessary for the definition of a concepts class; its arguments can be written directly into the con-

Table 2: The HasEqual concept in (a) C++0x and (b) Origin

(a)

```
concept HasEqual<typename T, typename U> {
    bool operator==(T, U);
}
```

(b)

```
template<typename T, typename U>
struct HasEqual {
    HasEqual() { auto p = constraints; }

    static void constraints(T x, U y) {
        SameType<decltype(x == y), bool>{};
    }

    typedef tuple<
        has_equal<T, U>,
        SameType<typename equal_result<T, U>::type, bool>
    > requirements;

    typedef typename concept_check<requirements>::type type;
    static constexpr bool value = type::value;
};
```

cept_check template. However, we prefer to list requirements separately from the metafunction that evaluates them.

The type and value members cause the template to describe an integral metafunction concept (Abrahams and Gurtovoy, 2005). This allows the concept class to be used like any other type trait. Writing HasEqual<X, Y>::value will yield true if an operator == is defined for the types of X and Y and its result is bool.

This technique requires a concept's requirements to be restated in two different ways. Whereas the constraint class aspect of the concept expresses requirements as use patterns, the metafunction aspect uses type traits. Although redundant, we find it easier to read and write requirements as use patterns and derive the type trait requirements from that more natural specification. In an earlier version of the library, we experimented with using only type traits to write the requirements and then statically asserting those to induce compiler errors. Although effective, it made their maintenance much more difficult, so we switched to using a combination of use patterns and type traits.

4.2.1. Explicit Concepts

Origin concepts are, by nature, automatic concepts. The constraint class and type trait features are designed (and required) to be evaluated at compile time in order to enforce constraints. However, the mechanism by which this is accomplished also permits us to emulate explicit concepts. In the Origin model, an explicit concept is one that is not automatically unsatisfiable. For example,

we define ForwardIterator as:

```
template<typename X>
struct ForwardIterator : InputIterator<X> {
   ForwardIterator() {
      static_assert(explicit_concept<X>::value, "Concept is explicit");
   }

   typedef tuple<explicit_concept<X>> requirements;
   typedef typename concept_check<requirements>::type type;
   static constexpr bool value = type::value;
};
```

The concept is made explicit because the explicit_concept type trait is always false. This has the effect of a) forcing static assertions whenever the constructor is instantiated and b) always returning false when evaluated as a type trait. The concept can be satisfied only by specializing the template for a specified type.

The Origin Concept library also encapsulates this behavior in a class template, Explicit, which can be used as a base class to simplify the definition of explicit concepts. This allows us to simplify the definition of explicit concepts:

```
template<typename X>
struct ForwardIterator : InputIterator<X>, Explicit
{ };
```

There are some technical issues related to its use when the concept participates in a concept hierarchy, which we discuss in Section 4.3.

This approach is not quite analogous to explicit refinement as described by Stroustrup (Stroustrup, 2009b). We consider the entire concept to be explicit, not just the conversion between the two. Because Origin is unable to emulate the semantics of explicit refinement, we cannot use the Origin library to effectively gauge the impact of that idea on library design.

4.2.2. Associated Operations

Unfortunately, there is no simple analog for the function declaration style of expressing requirements on associated operations that was proposed for C++0x (Gregor et al., 2006). The techniques employed here are based on the specification of use-patterns (Dos Reis and Stroustrup, 2006) both as part of constraint checking classes and within the definitions of SFINAE-based type traits. In turn, this creates a substantial burden for concept designers since they must implement the type traits required by their concepts and keep them in sync with the specification of the valid expressions. Specifying concepts that require only a handful of operations can be quite burdensome, and the concepts themselves can become quite complex. For example, a comparison of a partial specification of the C++0x Iterator concept to its emulated concept class is shown in Table 3.

Here, we only show operational requirements. We have omitted requirements on associated types and the results of various operations. In some respects, however, the application of use-patterns to specify constraints results in a more

Table 3: A partial listing of requirements for the Iterator concept in (a) C++0x (b) Origin

(a)

```
concept Iterator<typename X> : Semiregular<X> {
    typename reference;
    typename postincrement_result;
    reference operator*(X&);
    reference operator*(X&&);
    X& operator++(X&);
    postincrement_result operator++(X&, int);
}
```

(b)

```
template<typename X>
struct Iterator : Semiregular<X> {
    Semiregular() { auto p = constraints; }
    static void constraints(X iter) {
        *iter;
        iter++;
        SameType<decltype(++iter), Iter&>{};
    }
    typedef tuple<
        Semiregular<X>,
        has_dereference<X>,
        has_post_increment<X>,
        has_pre_increment<X>
        SameType<typename pre_increment_result<X>::type, T&>
    > requirements;
    typedef typename concept_check<requirements> type;
    static constexpr bool value = value::type;
};
```

concise listing of requirements. Here, for example, we only need to require a single dereference operator and not provide two signatures.

4.2.3. Associated Types

An associated type, A, is a type that is associated with another type, T, by being the result of an operation on T (e.g., postincrement_result in Iterator) or a part of T's interface. Such types may be associated directly with the class (e.g., Iter::value_type) or through a traits class (e.g., iterator_traits<Iter>::value_type). A *traits class* is a class template (and specializations) that is used to decouple the specification of associated types from the class with which they are associated. Traits classes are needed when the original class cannot be extended to define the additional information (Myers, 1995).

The role of associated types in our approach is limited. We only create aliases for types in order to constrain them. We do not attempt to emulate the proposed lookup rules or make them part of the interface of the concept. Unnamed associated types resulting from an operation are deduced using the

decltype operator. Named associated types are used in the same way that they are used for type traits (Alexandrescu, 2000) or traits classes (Myers, 1995). A partial listing of Iterator requirements for associated types and its corresponding C++0x syntax are given in Table 4.

Table 4: Associated type requirements for the Iterator concept as written in (a) C++0x and (b) Origin

(a)

```
concept Iterator<typename X> : Semiregular<X> {
    typename reference = X::reference;
    typename postincrement_result;
}
```

(b)

```
template<typename X>
struct Iterator : Semiregular<X> {
    Semiregular() { auto p = constraints; }
    static void constraints(X iter) {
        typedef typename iterator_traits<X>::reference reference;
        typedef decltype(iter++) postincrement_result;
    }
};
```

Here, the reference type is derived from iterator_traits, and postincrement_result is the result of the expression iter++. We note that reference could also be deduced from the expression *iter. Since the C++0x concept explicitly refers to that name, we follow suit and use the traits class.

4.2.4. Associated Requirements

Associated requirements are simply applications of nested constraints on either a template argument or associated type. In Origin, associated requirements must be written in both the constraints checking and metafunction components of the concept class.

Although the specification of requirements appears redundant, two distinct purposes are being served. Within the constraints function, the requirements are instantiated to force compiler errors. Within the requirements tuple, they are simply listed as evaluable type traits for later investigation. Note that the specification of associated requirements is otherwise no different than the use of concept classes to constrain the results of associated operations. These are shown in Table 5.

4.2.5. Refinement

In our approach, inheritance can be used to syntactically duplicate the notion of concept refinement but not semantically. In the C++0x concepts proposal, refinement defines a partial order on concepts that supports concept-based overloading. This is not the case in our library-based approach. In fact, in our

Table 5: Requirements on associated types for the Iterator concept in (a) C++0x (b) Origin

(a)

```
concept Iterator<typename X> : Semiregular<X> {
    requires MoveConstrutible<reference>;
    requires MoveConstrutible<postincrement_result>;
    requires HasDereference<postincrement_result>;
}
```

(b)

```
template<typename X>
struct Iterator : Iterator<X> {
    Iterator() { auto p = constraints; }
    static void constraints(X iter) {
        MoveConstructible<reference>{};
        MoveConstructble<postincrement_result>{};
        HasDereference<postincrement_result>{};
    }
    typedef tuple<
        Semiregular<X>,
        MoveConstructible<reference>,
        MoveConstructble<postincrement_result>,
        HasDereference<postincrement_result>
    > requirements;
};
```

approach there is no difference between using inheritance or composition to introduce additional requirements. It is simply a syntactic device. Two equivalent implementations of the Iterator class are shown in Table 6.

The use of inheritance can be problematic, especially when multiple concept classes are inherited, because they introduce ambiguous names. In such cases, we typically try to identify a "dominant" base concept and write the others as associated requirements. An alternative approach is to re-declare the appropriate typedefs or import the required names with using declarations.

4.2.6. Provisions

Some concepts can provide default implementations of required functions given a kernel of core operations. For example, given an == operator we can easily derive an implementation of !=. We call such operators *provisions*. From our perspective, provisions are not requirements; they are simply default implementations of required operations, and can (generally) be implemented using the standard enable_if idiom (Järvi et al., 2003). For example, we can provide a default implementation of != for any type satisfying the HasEqual requirements.

```
template <typename T, typename U>
inline auto operator!=(T const& x, U const& y)
    -> typename enable_if<HasEqual<T, U>::value, decltype(!(x == y))>::type
{
   return !(x==y);
```

Table 6: Equivalent requirements written using (a) inheritance and (b) composition

(a)

```
template<typename X>
struct Iterator : Semiregular<X> {
    // ... other requirements
    typedef tuple<Semiregular<X>, /* ... */ > requirements;
};
```

(b)

```
template<typename X>
struct Iterator {
    Iterator() { auto p = constraints; }
    static void constraints(X iter) {
        Semiregular<X>{};
        // ... other requirements
    }
    typedef tuple<Semiregular<X>, /* ... */ > requirements;
};
```

}

The operation can be defined in the global namespace to make it available to all types that satisfy the requirements of the operation. The enable_if restricts instantiation of the overload to only types modeling the HasEqual concept. The result type is given as the result of the returned expression. This preserves the result type of the operation for the template parameters. For example, if T and U are expression templates (Veldhuizen, 1995), then the result type is a new expression template. In cases where the a type has implemented its own operator !=, the partial ordering of overloads will select the more specialized implementation.

4.2.7. Axioms

An axiom describes an invariant (a semantic requirement) on types and is written in syntax similar to a function definition. It is important to note that, at best, these invariants can only be checked at runtime e.g., using assertions. There are also cases where it is not desirable to assert semantics. For example, asserting properties of an InputIterator may accidentally consume the input to an algorithm. Checking semantics at runtime can actually be the *cause* of program errors! Current C++ compilers cannot automatically verify the properties described by an axiom. To do so would require mechanics for reasoning about the meaning of programs: a requirement currently outside the scope of C++ program translation.

In our model, an axiom is simply a function template that evaluates a predicate. This allows runtime verification, if the user wants it. For example, the *reflexive*, *symmetric*, and *transitive* properties associated with the equality operator required by the EqualityComparable concept shown in Table 7:

The definitions are fairly similar, except the scope in which they are declared

Table 7: Semantic requirements defined as (a) axioms in C++0x and (b) predicate templates in Origin

(a)

```
concept EqualityComparable<typename T> {
    axiom Reflexive(T a) { a==a; }
    axiom Symmetric(T a, T b) { a==b => b == a; }
    axiom Transitive(T a, T b, T c) { a==b && b==c => a==c; }
}
```

(b)

```
template<typename T>
bool Eq_Reflexive(T a) { return a==a; }

template<typename T>
bool Eq_Symmetric(T a, T b) { if(a==b) return b==a; }

template<typename T>
bool Eq_Transitive(T a, T b, T c) { if(a==b && b==c) return a==c; }
```

and the use of the implication operator (=>). In Origin, axiomatic properties are not directly associated with concept classes in order to support reuse.

Axioms are associated with a concept by calling them within the body of a constraints function of a concept class. Although the axiom function is never actually executed, it is instantiated, and provides some additional static checking. The EqualityComparable concept is thus defined as:

```
template<typename T>
struct EqualityComparable {
   EqualityComparable() { auto p = constraints; }
   static void constraints(T x, T y, T z) {
      SameType<decltype(x == y), bool>{};
      Eq_Reflexive(x);
      Eq_Symmetric(x, y);
      Eq_Transitive(x, y, z);
};
```

Here, we omit any further checking of the result since the function is not intended to be evaluated, anyways. Simply writing the calls in this way should be sufficient to establish a documented semantic requirement. We further note that because axiom functions are evaluable function templates, they could easily be integrated into a test suite to evaluate invariants of models.

4.3. Modeling Concepts

Most of the concepts in the Origin Concept library are checked automatically. Explicit concepts are still required to disambiguate models that cannot be syntactically differentiated (e.g., InputIterator and ForwardIterator). Concept maps in the Origin Concept library are implemented as class template specializations of a concept class. For example, we can provide a concept map that explicitly satisfies the requirements of ForwardIterator for pointers by writing:

```
template<typename T>
struct ForwardIterator<T*> : InputIterator<T*>
{ };
```

This declaration effectively states that all pointer types (T*) are valid ForwardIterators if they are valid InputIterators. Note that this also satisfies queries asking if pointers are RandomAccessIterators. In cases where the explicit concept has no refinements (base concepts), the Origin Concept library provides the base class Model, which never fails a concept check.

In some cases, concepts can be defined to express overlapping syntactic requirements. Although we have not generally found this to be problematic, it can lead to ambiguities if functions are overloaded on those overlapping requirements. In such cases, concepts maps must be used to disambiguate the lookup. One way to do this is to negatively assert a concept using a negative concept map. These are briefly discussed by Stroustrup (Stroustrup, 2009b). Suppose, for example, that we define a function, sort, on Watch types (e.g., the stopwatch model) that orders split times by duration, and furthermore that Watches have a begin and end operation, just like Ranges. Calling sort(x) is ambiguous if both concepts are automatically checked; it could mean sort an iterator Range or sort split times in a Watch. Negating one of the concepts can disambiguate the problem. This is can be done by defining a concept map that derives from NegatedModel rather than Model.

```
template<typename Duration>
struct Range<stopwatch<Duration>> : NegatedModel
{ };
```

The NegatedModel behaves like the Explicit concept described previously in that it always fails concept checks. However, it is unclear to what extent this feature would actually be useful in real generic libraries. It is even less clear how often these kinds of problems arise. We opt to address the issue here in order to demonstrate the viability of our approach to address this problem.

The use of template specialization to create concept maps can create "gaps" in the concept checking features of the library. For example, a programmer might erroneously declare this:

```
template<typename Iter>
struct Forward_iterator<istream_iterator<T, Char, Traits>> : Model
{ };
```

An istream_iterator is never a Forward_iterator. Concept maps can wrongfully claim that a type models a concept, even when it does not. In Origin, this would not be caught by the compiler and could ultimately lead to a serious program error. This is a known limitation of the approach. Because the checked requirements are defined within the concept class, concept map specializations can simply bypass those syntactic checks. We can, however, recommend strategies to avoid this problem.

One approach is to avoid writing statically checked requirements in explicit concepts. An explicit concept with statically checkable requirements can gener-

ally be refactored into two concepts: an automatic concept including the statically checkable requirements and an explicit concept containing the semantic requirements. This forces the user to explicitly state the semantic requirements of a type while the static properties are explicitly checked. Alternatively, the automatic component could be required within the concept map to explicitly check static conformance.

4.4. Constraining Templates

Since C++ lacks syntax for imposing type constraints on templates, designers of generic libraries must work within the confines of the existing language mechanics to provide these features. The concept classes of the Origin Concept library address two aspects of constraining templates: statically asserting concept requirements and supporting concept-based overloading.

4.4.1. Asserting Requirements

Asserting requirements always results in compilation failure if requirements are not satisfied. This is achieved using the constraint class aspect of a concept class. Specifically, instantiating a concept class in such a way that the default constructor is also instantiated will cause the constraints to be evaluated. For example, in a function template we can create a temporary object by explicitly invoking the default constructor. Within a class template, this is done by deriving from the required concept class(es). These techniques are exactly the same as those discussed in Section 4.1.1 We note that neither technique affects the performance or memory profiles of the constrained program; as the compiler's optimizer will remove the unused temporaries and space required by the empty base class.

There are some cases however, where constraints might be enforced in a context where such declarations are neither permissible nor desired. For example, a programmer may want to constrain a derived associated type (typedef) within a class definition. This can be done by statically asserting the concept class as a metafunction predicate. For example:

```
template<typename Vertex, typename Edge, typename Selector>
class adjacency_list {
   typedef typename Selector<Vertex>::type VertexList;
   static_assert(Container<VertexList>::value, "VertexList is not a Container");
   // ...
};
```

This allows a data structure to issue very specific errors messages. Note that the emitted message is specific to the usage context rather than the properties of the type. Checking the Container concept as a function or base class constraint might yield less informative error messages. We discuss generated error messages in Section 4.5.

4.4.2. Overloading with Concepts

Concept-based overloading is traditionally accomplished using two techniques. Tag dispatch relies on existing overload rules to select an overload based on the type of a tag class (an empty class). When organized into tag class hierarchies, tag dispatch supports overloading based on the subtype order. This technique is the basis of fundamental operations on Iterator types and functions such as advance, and distance (Austern, 1998). Concept-controlled polymorphism allows library designers to restrict overload candidates based on properties of types (Järvi et al., 2003). While our emulation approach favors the use of concept-controlled polymorphism via the enable_if mechanism, we do not exclude the use of tag dispatch. That technique simply requires more implementation support than is currently provided.

Because Origin's concept classes are evaluable as metafunction predicates, they can be used with the enable_if function. We also can use this technique to build conditions that emulate the overloading properties of tag dispatch by defining requirements that partition types by their requirements. For example, we can implement the advance function as:

```
template<typename Iter>
void advance(Iter& i, int n,
             typename enable_if<
               InputIterator<Iter>::value && !BidirectionalIterator<Iter>::value
             >::type* = nullptr);

template<typename Iter>
void advance(Iter& i, int n,
             typename enable_if<
               BidirectionalIterator <Iter>::value &&
               !RandomAccessIterator<Iter>::value
             >::type* = nullptr);

template<typename Iter>
void advance(Iter& i, int n,
             typename enable_if<
               RandomAccessIterator<Iter>::value
             >::type* = nullptr);
```

Admittedly, the syntax leaves much to be desired. However, we know of no better library-based alternatives. Even tag dispatch requires its own infrastructure: tag classes must be defined and associated with types, and the dispatching algorithm must delegate to one of several dispatch targets. The use of enable_if requires somewhat less infrastructure at the expense of open extensibility.

We further note that this particular approach, constructing predicate bounds based on satisfied requirements, defines a closed system. It is not possible to extend the definition of advance to new iterator concepts without changing the constraints on the most specialized algorithm. Tag dispatch allows open definitions. We discuss issues related to the integration of tag dispatch mechanisms into the Origin Concept library in Section 6.

4.4.3. Archetypes

Although the Origin Concept library does not include archetypes, we feel that they warrant discussion. Archetypes are used to ensure that a template definition does not rely on types or operations that are not specified by its requirements. An *archetype* is a class that exactly satisfies the requirements of a concept, providing no other operations or associated types other than what is required. The behavior of the associated operations is undefined. Associated types are also archetypes determined by associated requirements on those types.

For example, an archetype for LessThanComprable could be defined as:

```
template<typename Base = null_arch<>>
class less_than_comparable_arch {
   boolean_archetype operator<(const less_than_comparable_arch& x) const {
      return static_object<boolean_arch>::get();
   }
};
```

The template uses the Curiously Recurring Template Pattern (Coplien, 1995) to support archetype composition, defaulting to null_archetype. The null_archetype class prevents default and copy construction by hiding (or deleting) those constructors. The only operation exposed is the operator <, and it returns boolean_archetype, a behavior-less representation of a Boolean type. The get function of the static_object class is conceptually similar to the declval function. It will, however, fail in dramatic fashion if the compiled program is actually executed.

The Boost Concept Check Library [6] (Siek and Lumsdaine, 2000) provides corresponding archetype classes for every concept it defines. These are used in the test suite to instantiate generic algorithms and data structures. If the instantiation uses an operation or type that is not directly provided by the archetype class, compilation fails. The archetype can be used to check the max algorithm, for example:

```
auto const& x = static_object<has_less_archetype<>>::get();
auto const& y = static_object<has_less_archetype<>>::get();
max(x, y);
```

Here, the objects x and y are objects of HasLess' corresponding archetype. If the max algorithm uses the greater than (>) operator instead of the less than < operator, compilation will fail because x and y cannot be compared with >. Compilation could also fail if the arguments and return value were passed by value because the has_less_archetype is non-copyable; the HasLess concept does not require its arguments to be copyable, and so the corresponding archetype must explicitly disable the functionality.

The Origin Concept library does not currently define archetypes for its associated concepts. Such a feature would be useful for helping to validate design experiments, but it was not considered a critical requirement for our experiments.

4.5. Error Messages

The generation of error messages is primarily handled by the concept class' constraint checking features. For example, trying to instantiate the find algorithm over a pair of ints might yield the following diagnostics:

```
In static void InputIterator<Iter>::constraints(Iter) [with Iter = int]':
   instantiated from 'InputIterator<Iter>::InputIterator() [with Iter = int]'
   instantiated from 'void find(Iter, Iter, const T&) [with Iter = int, T = int]'
error: invalid type argument of unary '*' (have 'int')
```

Unfortunately, the compiler diagnostics are not dramatically improved. The error is still written in terms of the failed substitution rather than the conceptual requirement. However, the error message does cite the concept class in which the error occurred. We hope that programmers can use this information as a beacon to help guide their search for the error. The full template instantiation stack is also printed. Here, the stack is shallow. In deeply nested template errors, this will not be the case. In order to avoid deeply nested template errors, template definitions could be constrained with the full set of requirements for all nested calls and used data structures.

The error messages produced when checking valid expressions are generated by the compiler. More meaningful error messages could be generated using static assertions. If we statically asserted each type trait requirement in the constructor of each concept class, very specific and meaningful error messages could be generated. For example, the HasEqual concept could be defined thusly:

```
template<typename T, typename U>
struct HasEqual {
   HasEqual() {
      static_assert(has_equal<T, U>::value, "no operator== for T and U");
      static_assert(is_same<typename equal_result<T, U>::type, bool>::value,
                    "operator== for T and U does not return bool");
   }
};
```

We experimented with this approach in an early version of the library, but eventually adopted the use-pattern technique. Although the error messages generated with use patterns are less clear, the requirements themselves are easier to read and write.

5. Implementation

The Origin Libraries is a collection of generic algorithms and data structures written in the emerging C++11 programming language. Origin is comprised of a two major components: core libraries and application libraries. The core libraries are essentially those described in this paper; they include:

- *meta*—The metaprogramming library provides core support for nearly every generic library in Origin including facilities to work with type sequences.

- *traits*—The traits library is a collection of type traits like those described in Section 4.1.2. This library extends the metaprogramming library and makes the concept emulation library possible.

- *concepts*—The concepts library implements provides the facilities to implement concept classes and includes a number of commonly used concepts (e.g., SameType).

The application libraries implement provide generic data structures and algorithms, and the concepts that describe their abstractions. Existing libraries include:

- *iterator*—The iterator library provides iterator abstractions

- *range*—The range library abstracts iterator ranges (pairs of iterators)

- *data*—Data structures including as dynarrays and heaps.

- *graph*—The graph library library provides graph data structures and algorithms.

- *sandbox*—A collection of experimental libraries that contain nascent implementations of libraries in the mathematical, topological, and statistical domains.

The Origin libraries are written using the emerging ISO C++11 programming language standard. We do not rely on features specific to any compiler and so the library should be portable. However, there is (at the time of writing) no consistent industry-wide compiler support for the range of C++11 programming language features required to actually compile the Origin. Origin is reliably built against GCC's most recent Subversion revision.

We also note that Origin is also a moving target; the Origin libraries are often refactored to use new features as they become available or to incorporate new ideas. the intent of the project is to support experimentation with generic programming and generic library design. In this paper, we are reporting on an experiment involving the emulation C++0x concepts in an effort to discover related principles and problems and ultimately support decisions about language design.

6. Discussion

One of the most successful features of the design of the Origin Concept implementation is that our approach emphasizes the predicate aspect of concepts: a type models a concept, or it does not. This is similar to the definition used by Stepanov and McJones in *Elements of Programming* (Stepanov and McJones, 2009). In this work, the authors define concepts mathematically as the conjunction of syntactic and semantic requirements on types. They do not use the concept syntax proposed for C++0x, nor do they rely on notions of concept refinement, concepts maps, or archetypes. Our emulation of concepts aligns well

with those used by Stepanov and McJones. Origin's automatically checked concept classes provide a facility for implementing the concepts defined in *Elements of Programming*. We plan to further evaluate the applicability of the library by implementing the concepts and algorithms presented in their book. We hope that the experiment will provide further insight into the design of concepts and concept systems.

In this section, we discuss a number issues related to the design and implementation of the library. Specifically, we describe various approaches considered to emulate different language features and design techniques for defining concepts in Origin. The intent of this discussion is to demonstrate the applicability of the library to a broad set of design problems, especially those relevant to the design of language features for concepts.

6.1. Concept Tags

In Section 4.4.2, we noted that tag dispatch can be used to support an open overloading mechanism. The underlying principle of this technique is the mapping of types to tag classes in order to induce subtype-based ordering on candidate overloads. For example:

```
template<typename Iter>
int distance(Iter first, Iter last, input_iterator_tag);

template<typename Iter>
int distance(Iter first, Iter last, random_access_iterator_tag);

template<typename Iter>
int distance(Iter first, Iter last)
{
    distance(first, last, typename Iter::iterator_category{});
}
```

Here, the input_iterator_tag and random_access_iterator_tag are empty classes related through inheritance (the former is a base class of the latter). The distance algorithm dispatches to an overload based on the iterator's associated iterator_category type. If Iter's category is anything other than random_access_iterator_tag, the first overload is called. Otherwise, the second overload is called.

The tag classes in this system represent iterator concepts, and their organization by inheritance hierarchy approximates the notion of refinement. In early iterations of the Origin concept library, we had considered generalizing the implementation of concept checking to use tag classes. In other words, the requirement InputIterator<X> would internally check to see if X's iterator category was input_iterator_tag.

However, we found that the approach is not feasible for large numbers of orthogonal concept hierarchies, such as those found in the C++0x Draft Standard (Becker, 2009). For example, an int models concepts in the following hierarchies or domains: linguistic (ObjectType), operations (HasPlus), comparison (EqualityComparable), construction (DefualtConstructible), regular (Regular),

and numeric (IntegralLike). Obviously, these are not the only concepts modeled by int; this is just a small sample. Using tag classes as the only basis for checking type constraints would require that we exhaustively enumerate every knowable property of int and even its related types (int&, int*, const int, etc) using tag classes and concept maps. Clearly, the approach does not scale, so we rejected it as the exclusive basis for our concept emulation approach.

6.2. Casual Models

One of the side effects of adopting automatic concepts as the default approach to concept checking is what we refer to as "casual modeling". We say that a type casually models a concept if it unintentionally satisfies the concept's syntactic but (perhaps) not semantic requirements. There are often two reasons why this might the case, there are semantic differences or overlapping concepts.

6.2.1. Semantic Differences

A concept may differ semantically but not syntactically from a base concept. This is exemplified by the InputIterator and ForwardIterator concepts. A ForwardIterator is an InputIterator if its template argument also satisfies the semantic MultiPass axiom. With purely automatic checking, this will lead to a scenario in which an InputIterator is misclassified as ForwardIterator. Any multi-pass actions on the underlying iterator will consume the underlying state, making the algorithm incorrect, likely leading to serious errors. The most well known instance of this problem is the definition of range constructors of vectors illustrated in Section 3.5.

If an InputIterator, say istream_iterator, is misclassified as a ForwardIterator due to automatic concept checking, the vector will be constructed with n *uninitialized* objects! An initial call to distance computes the space that should be reserved for the objects that will be copied into it. Unfortunately, this call also consumes the elements in the iterated istream. After calling distance, no more objects can be read from the underlying stream and so no objects are copied into the reserved memory, leaving it uninitialized. From this, we conclude that any language design that supports only automatic concepts is incompatible with the requirements of real world programs.

We see two immediate solutions to this problem. First, we could make the ForwardIterator an explicit concept, requiring the programmer to supply a concept map. This is the solution discussed in Section 4.3. Second, we could use some other mechanism to evaluate the semantic requirement, effectively making the concept statically differentiable from InputIterator. For example, we might rely on the iterator's iterator_category to support static checking of ForwardIterators. In other words, an InputIterator is a ForwardIterator if its iterator_category is convertible to forward_iterator_tag. Although this works well in this case, the approach does not necessarily scale since not all types have associated tag classes, nor should they for the reasons described in Section 6.1.

6.2.2. Overlapping Concepts

There are cases, alluded to in Sections 3.5 and 4.3, where the specification of overlapping concepts can lead to ambiguities. For example, both the Range and Container concepts in the C++0x Draft Standard require operations begin and end although the concepts themselves are unrelated. We note that all Container types implicitly model the Range concept, assuming both are automatically checked. Suppose we declare a function print as:

```
template<Range R> void print(R const& r);
template<Container C> void print(C const& c);
```

Calling print on a type such as iterator_range<Iter> will not result in ambiguous lookup since it is not possible for the iterator_range template to satisfy the requirements of Container (it does not define size, empty, front, etc). The converse is untrue. Container types such as list<T> satisfy the requirements of both overloads resulting in an ambiguous lookup.

One solution, the one that we take with Origin, is to constrain the Range-based print function differently. Specifically, we would write its constraints as Range<T> && !Container<T>. In other words, the overload is viable for any type that is a Range, but not a Container.

A related solution would be to define a Container as a refinement of Range so that the overloads can be ordered with respect to that refinement. However, this solution implies that Container and Range are both syntactically and semantically inherently related, which may not be an appropriate classification of concepts. The emulation of this solution could use tag dispatch (for an open system) or the predicates previously stated (for a closed system).

A third solution would be to make either concept explicit. However, we see this solution as tailoring the design of a concept to a constraints problem. We might also use negative concept maps to assert that a particular type is not a Range or Container. Of these two options, making Container explicit is the better choice since failing to negate an explicit concept can lead to runtime errors. Failing to affirm an explicit concept should, at best, result in loss of performance.

Stroustrup discusses the disambiguation of overlapping subsets of requirements (Stroustrup, 2009b). One proposal is to use a concept map to non-intrusively state a relation between e.g., Range and Container (i.e., all Containers are Ranges). Our emulation approach does not support this type of declaration, and so it is not possible to evaluate this particular solution except hypothetically.

6.3. Provisions and Contextual Models

In an early design of the Origin Concept library, we had attempted to approximate the behavior of provisions by implementing them as unconstrained templates in a namespace associated with the concept. For example, the operator != was defined for EqualityComparable as:

```
namespace EqualityComparable_ {
```

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

An algorithm requiring EqualityComparable can import the namespace (using namespace EqualityComparable) to make the operation available in the constrained scope. Although we believe that this technique faithfully approximates the intended behavior of the C++0x concept proposal, we rejected the implementation for two reasons. First, it is too cumbersome for library designers. In templates with non-trivial requirements, any number of namespaces might be imported, which can dramatically increase the potential for ambiguous lookup.

The second reason is more fundamental. Suppose, in C++0x we define a class number and we want it to be EqualityComparable. To do so, we would only need to implement ==, and != would be provided by the default implementation. Or is it? Consider

```
int main() {
  number a = 0, b = 1;
  assert(a != b); // Error: no such !=
}
```

The reason that this results in an error is that the context is unconstrained; the EqualityComparable concept has not brought a defining of != into scope. This is the same behavior we can expect by emulating the lookup using associated namespaces. The implication is that number is only EqualityComparable in appropriately constrained contexts and not otherwise.

This defines an inconsistent model for end users of concepts. If a type is known to model a concept, then all of its required and provided operations should also be available in any context. This is the reason that we opted to use constrained templates (via enable_if) to provide default definitions of those operations. We view requirements as being wholly distinct from their default implementations. It is not clear what impact this observation has on the design of concept systems. Certainly, it is a design technique that deserves further experimentation and analysis.

6.4. Shallow Requirements, Deep Errors

In Section 4.5, we described the error messages generated by the Origin Concept library. When a concept error is reported using this library, the entire template instantiation stack is included in the compiler diagnostics. If the concept error occurs within a deep nesting of template instantiations (e.g., more than, say, 5 levels), the root cause of failure can be difficult to find in the output.

Deeply nested template errors indicate that requirements have not been fully specified in the top-level template. For a simple two-level instantiation stack, consider the following program written using the Origin concepts:

```
template<typename Seq>
```

```
void print(Seq s, enable_if<Sequence<Seq>::value>::type* = nullptr) {
   Printable<Seq::value_type>{};
   for(const auto& x : s)
      print(x)
}

template<typename Seq>
void reverse(Seq s) {
   Sequence<Seq>{};
   print(s);
}
```

Here, a generic algorithm operating on a Sequence prints that sequence. The print function also requires that the Sequence's value type be Printable. If reverse is instantiated over a sequence with non-printable value types, an error would originate at the Printable requirement in the print function, two levels deep. It is easy to see how such errors easily occur at deeper levels, and the Origin Concept library does not guard against this.

Note that the error could be caught within the reverse function by adding the Printable requirement to that. This allows the error to be caught within the body of the top-level template. Catching all conceptual errors by fully specifying requirements in a top-level template (i.e., shallow requirements) may lead to more readable error message since nested templates may not be instantiated before the concept check fails.

However, there is an obvious problem with this strategy. No reasonable implementation of reverse should require all input types to be Printable. The call to print may have been added by a programmer to assist in debugging. It is not a general requirement of the algorithm.

In C++0x, the original call to print would not have been allowed. The separate checking of template definitions would require the Printable requirement to be introduced at the top-level. As a work-around, a late_check block could be introduced to allow calls to unconstrained or orthogonally constrained functions. These are the same issues related to usability issues of constrained and unconstrained templates discussed by Stroustrup (Stroustrup, 2009b). Our concern is that imposing a model of strict checking will make it more difficult to maintain generic algorithms and data structures, and we wonder if a better approach might be to use language features similar to final in Java or sealed in C# to enforce strict checking rather than late_check to escape it. A more empirical analysis of the trade-offs is required.

7. Conclusions

In this work, we presented a library-based approach to concept emulation and described the sets of features that can be approximated. The choice of using emulation (as opposed to defining language extensions or building a pre-processor) allows us to experiment directly with the underlying mechanics of an intended language. We used the Origin library to define many of the concepts

found in the C++0x draft standard in order to evaluate the applicability of our implementation to this and similar tasks. We have also used the language to reconstruct a number of usability issues discussed by Stroustrup (Stroustrup, 2009b), allowing us to reason about them in more concrete terms.

At the heart of many of these issues is the distinction between explicit and automatic concepts. In essence, automatically checked concepts introduce a number of problems in concept checking, overload resolution, and even basic interface properties. Explicit concepts have none of these problems, but require a programmer to list the complete set of concepts modeled by every type. We conjecture that the exclusive use of explicit concepts to define and check type properties is unlikely to scale when considering the extensive amount of legacy source code that would have to be modified to work with those concepts.

The results of the work demonstrate the viability of a library-based emulation approach for empirically investigating issues related to language design. The problems and trade-offs that can be addressed by the library are issues that must be addressed by any proposed language extension.

Acknowledgements

We would like to acknowledge Michael Lopez and Brian Bartman for their contributions to discussion about the design of the library and the techniques used. We also like to thank Bjarne Stroustrup for providing historical perspectives on concepts and emulation approaches and his insight into usability issues related to concepts.

References

Abrahams, D., Gurtovoy, A., 2005. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. C++ In-Depth Series. Addison Wesley.

Alexandrescu, A., Oct 2000. <generic programming>: Mapping between types and values. Dr. Dobb's: The World of Software Development.

Alexandrescu, A., 2001. Modern C++ Design: Generic Programming and Design Patterns Applied. C++ In-Depth. Addison Wesley.

Austern, M., 1998. Generic Programming and the STL: Using and Extending the C++ Standard Template Library, 7th Edition. Addison-Welsey Longman, Boston, Massachusetts.

Becker, P., 2009. Working draft, standard for the programming language C++. Tech. rep., ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++.

Coplien, J., 1995. Curiously recurring template patterns. C++ Report 7 (2), 24–27.

David, V., Haveraaen, M., 2009. Concepts as syntactic sugar. In: 9th International Conference on Source Code Analysis and Manipulation (SCAM'09). Edmonton, Canada, pp. 147–156.

Dos Reis, G., Stroustrup, B., 2006. Specifying C++ concepts. In: 33rd Symposium on Principles of Programming Languages (POPL'06). Charleston, South Carolina, pp. 295–308.

Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J., 2003. A comparative study of language support for generic programming. In: 18th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03). Anaheim, California, pp. 115–134.

Gregor, D., 2009. ConceptGCC.
URL http://www.generic-programming.org/software/ConceptGCC/

Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A., 2006. Concepts: Linguistic support for generic programming in C++. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systemls, Languages, and Applications (OOPSLA'06). ACM Press, Portland, Oregon, pp. 291–310.

Järvi, J., Willcock, J., Lumsdaine, A., 2003. Concept-controlled polymorphism. In: 2nd International Conference on Generative Programming and Component Engineering (GPCE'03). Erfurt, Germany, pp. 228–244.

McNamara, B., Smaragdakis, Y., Oct 10 2000. Static interfaces in C++. In: 1st Workshop on C++ Template Metaprogramming. pp. 1–12.

Musser, D., Stepanov, A., 1994. Algorithm-oriented generic libraries. Software: Practice and Experience 24 (7), 623–642.

Myers, N., Jun 1995. Traits: A new and useful template technique. C++ Report 7 (5), 32–35.

Siek, J., Gregor, D., Garcia, R., Willcock, J., Järvi, J., Lumsdaine, A., 2005. Concepts for C++0x. Tech. rep., ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++.

Siek, J., Lumsdaine, A., Oct 10 2000. Concept checking: Binding parametric polymorphism in C++. In: 1st Workshop on C++ Template Programming. Erfurt, Germany, pp. 1–12.

Siek, J., Lumsdaine, A., 2005. Language requirements for large-scale generic libraries. In: 4th International Conference on Generative Programming and Component Engineering (GPCE'05). ACM Press, Tallinn, Estonia, pp. 405–421.

Stepanov, A., McJones, P., 2009. Elements of Programming. Addison Wesley, Boston, Massachusetts.

Stroustrup, B., 1994. The Design and Evolution of C++. Addison-Wesley.

Stroustrup, B., Jul 22 2009a. The C++0x 'remove concepts' decision. Dr. Dobb's.

Stroustrup, B., 2009b. Simplifying the use of concepts. Tech. rep., ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++.

Stroustrup, B., Dos Reis, G., 2005. A concept design (rev. 1). Tech. rep., ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++.

Veldhuizen, T., 1995. Expression templates. C++ Report 7 (5), 26–31.

Zólyomi, I., Porkoláb, Z., 2004. Towards a general template introspection library. In: 3rd International Conference on Generative Programming and Component Engineering (GPCE'04). Vancouver, Canada, pp. 266–282.