Static interfaces in C++

Brian McNamara and Yannis Smaragdakis

Georgia Institute of Technology
lorgon@cc.gatech.edu, yannis@cc.gatech.edu

Abstract

We present an extensible framework for defining and using "static interfaces" in C++. Static interfaces are especially useful as constraints on template parameters. That is, in addition to the usual `template <class T>`, template definitions can specify that `T` "isa" `Foo`, for some static interface named `Foo`. These "isa-constraints" can be based on either inheritance (named conformance: `T` publicly inherits `Foo`), members (structural conformance: `T` has these member functions with these signatures), or both. The constraint mechanism imposes no space or time overheads at runtime; `virtual` functions are conspicuously absent from our framework.

We demonstrate two key utilities of static interfaces. First, constraints enable better error messages with template code. By applying static interfaces as constraints, instantiating a template with the wrong type is an error that can be caught at the instantiation point, rather than later (typically in the bowels of the implementation). Authors of template classes and template functions can also dispatch "custom error messages" to report named constraint violations by clients, making debugging easier. We show examples of the improvement of error messages when constraints are applied to STL code.

Second, constraints enable automatic compile-time dispatch of different implementations of class or function templates based on the named conformance properties of the template types. For example, `Set<T>` can be written to automatically choose the most efficient implementation: use a hashtable implementation if "`T` isa `Hashable`", or else a binary search tree if "`T` isa `LessThanComparable`", or else a linked-list if merely "`T` isa `EqualityComparable`". This dispatch can be completely hidden from clients of `Set`, who just use `Set<T>` as usual.

1. Introducing static interfaces

We begin by demonstrating how static interfaces could be useful, and then show how to emulate them in C++.

1.1 Motivation

To introduce the idea of "static interfaces", we shall first consider an example using traditional "dynamic interfaces"--that is, abstract classes:

```
struct Printable {
   virtual ostream& print_on( ostream& o ) const =0;
};
struct PFoo : public Printable {
   ostream& print_on( ostream& o ) const {
      o << "I am a PFoo" << endl;
      return o;
   }
};
```

Here "`PFoo` isa `Printable`" in the traditional sense of "isa". We have dynamic polymorphism; a `Printable` variable can be bound at runtime to any kind of (concrete) `Printable` object. Both named and structural conformance are enforced by the compiler: inheritance is an explicit mechanism for declaring the intent to be a subtype (named conformance), and the pure virtual function implies that concrete subclasses must define an operation with that signature (structural conformance). These conformance guarantees enable users to write functions like

```
ostream& operator<<( ostream& o, Printable& p ) {
   p.print_on(o);
```

```
      return o;
   }
```

so that "`std::cout << p`" works for any `Printable` object `p`.

The problem with this mechanism for expressing interfaces is that it is sometimes overkill. Virtual functions are a good way to express interfaces when we want dynamic polymorphism. But sometimes we only need static polymorphism. In these cases, interfaces based on abstract classes introduce much needless inefficiency. First, they add a vtable to the overhead of each instance of concrete objects (a space penalty). Second, they introduce a point of indirection in calls to methods in the interface (a runtime penalty). Finally, `virtual` calls are unlikely to be inlined (an optimization penalty).

As a result, libraries that exclusively use templates to achieve polymorphism (static polymorphism) usually avoid `virtual` altogether. The STL is one common example. However, there are no explicit language constructs to express "static interfaces". The only way to say, for example, that a type `T` "isa" `Printable` and that it supports the `print_on()` method with a particular function signature is to use abstract classes as described above. Thus, when templates rely on such interface constraints being met by the template type, the constraints are typically left implicit. At best, clients of template code may find constraints on template parameters in the documentation.

SGI's STL documentation [SGI] does an excellent job with template constraints; indeed, they go as far as dubbing such constraints "concepts", and their documentation describes "concept hierarchies". Each concept informally describes a particular interface requirement for a class. In order to use the STL, one must instantiate templates with types that are "models" of particular concepts like `EqualityComparable`, `LessThanComparable`, `ForwardIterator`, `RandomAccessIterator`, `CopyConstructable`, etc. These concepts exist explicitly only in the documentation; in the STL code they lie implicit. (Actually, SGI's own STL implementation now includes a kind of concept checking that is similar to the method we shall eventually describe at the beginning of Section 3.1.)

Most users of the STL are aware of concepts--it is hard to use the STL effectively if one has no idea of what a `ForwardIterator` is, for example. Concepts are important to understanding the whole framework of the STL, despite the fact that these concepts are not directly represented in C++. There are also meaningful *relationships* among concepts (e.g. a `RandomAccessIterator` "isa" `ForwardIterator`) which can only be expressed in documentation, as C++ has no explicit mechanisms to define concepts, much less describe the relationships among them.

The problems with leaving concepts implicit in the code are numerous. First, detection of improper template instantiations comes late (if at all), and the error messages are often cryptic and verbose. Second, there is no obvious way to determine what "concept constraints" there are for a template parameter without examining the implementation of the template extremely carefully. This makes using the template difficult and error-prone (unless there is copious documentation). Finally, there is no way to have the code "reason" about its own concept-constraints at compile-time (since the concepts are absent from the code).

In short, while C++ provides a nice mechanism for dynamic polymorphism, it provides no analogous mechanism for static polymorphism. Abstract classes allow users to specify constraints on parameters to *functions*, which may be bound to different *objects* at *run-time*. However there is no mechanism to specify constraints on parameters to *templates*, which may be bound to different *types* at *compile-time*. As a result, template code either must leave the constraints implicit, or make clever use abstract class hierarchies (which makes the performance suffer dramatically). However, as we shall demonstrate, static interfaces can give us the benefits of both approaches at once.

1.2 Imagining new language constructs

Let us imagine some new language constructs which let us explicitly express concepts. Consider a template function that sorts a vector. We can imagine writing:

```
template <class T> interface LessThanComparable {
   bool operator<( const T& ) const;
};

struct Foo models LessThanComparable<Foo> {
   bool operator<( const Foo& ) const { ... }
   // other members
};

template <class T isa LessThanComparable<T> >
// sort the vector with "operator<" as the comparator
void Sort( vector<T>& v ) { ... }
```

Note that there are three new keywords. **`interface`** lets us declare a static interface, which has a name and a structure, and which imposes no `virtual` overhead on the types that model it. **`models`** lets us declare that a type models the interface; the compiler will enforce that the type has the right methods. **`isa`** lets us express constraints on types; here, `Sort()` can only be called for `vector<T>` types where `T` is a model of `LessThanComparable<T>`.

This cleanly expresses what we would like to do. It is similar to the idea of "constrained generics" found in some languages (like GJ [BOSW98]), only without the ability to do dynamic polymorphism (and therefore without the associated performance penalties). It turns out we can do something very close to this using standard C++.

1.3 Emulating Static Interfaces in C++

Here is how we express a **static interface** (a concept) in our framework:

```
template <class T> struct LessThanComparable {
   MAKE_TRAITS;   // a macro (explained later)
   template<class Self> static void check_structural() {
      bool (Self::*x)(const T&) const = &Self::operator<;
      (void) x;   // suppress "unused variable" warning
   }
protected:
   ~LessThanComparable() {}
};
```

Note that we encode the structural conformance check as a template member function (which will be explicitly instantiated elsewhere, with `Self` bound to the concrete type being considered) to ensure that types conform structurally. This function takes pointers to the desired members to ensure that they exist. The protected destructor prevents anyone from creating direct instances of this class (but allows subclasses to be instantiated, which will be important in a moment). The `MAKE_TRAITS` line is a short macro for defining an associated traits class; its importance and use will be described in Section 2.1.

We use public inheritance to specify that a type **conforms** to a particular static interface (that is, the type *models* a concept):

```
struct Foo : public LessThanComparable<Foo> {
   bool operator<( const Foo& ) const { ... }
   // whatever other stuff
};
```

Then we use the **`StaticIsA`** construct to determine if a type conforms to a static interface. (More details of the implementation of `StaticIsA` will be explained later.) The expression

```
StaticIsA< T, SI >::valid
```

is a boolean value computed at compile-time that tells us if a type `T` conforms to a static interface `SI`. In other words, the value is `true` iff "`T isa SI`" (under the meaning of `isa` we described in Section 1.2). Thus, for example, in `Sort()`, we can use

```
StaticIsA< T, LessThanComparable<T> >::valid
```

to determine if a particular instantiation of the template function is ok. Since the value of `StaticIsA<T,SI>::valid` is a compile-time boolean value, we can use template specialization to choose different alternatives for the template at compile-time, based on `T`'s conformance.

2. Applications

In this section we show two useful applications of `StaticIsA`: custom error messages and selective implementation dispatch.

2.1 Using `StaticIsA` to create understandable error messages

Now that we have seen the idea behind `StaticIsA`, let us put it to use. Often a violation of the "concept constraints" for template arguments causes the compiler to emit tons of seemingly useless error messages. For example, with g++2.95.2 (which comes with an old SGI library), trying to `std::sort()` a collection of `NLC`s (where `NLC` is a type that does *not* support `operator<`) causes the compiler to report *fifty-seven* huge lines of error messages for one innocent-looking line of code. To emphasize the point, realize that this tiny (complete) program

```
#include <algorithm>
struct NLC {}; // Not LessThanComparable
int main() {
   NLC a[5];
   std::sort( a, a+5 );
}
```

compiled with g++ (without any command-line options to turn on extra warnings) produces a stream of error messages beginning with

```
/include/stl_heap.h: In function `void __adjust_heap<NLC *, int, NLC>(NLC *, int, int, NLC)':
/include/stl_heap.h:214:  instantiated from `__make_heap<NLC *,NLC,ptrdiff_t>(NLC *, NLC *, NLC *,
                                                                              ptrdiff_t *)'
/include/stl_heap.h:225:  instantiated from `make_heap<NLC *>(NLC *, NLC *)'
/include/stl_algo.h:1562:  instantiated from `__partial_sort<NLC *, NLC>(NLC *, NLC *, NLC *,
                                                                         NLC *)'
/include/stl_algo.h:1574:  instantiated from `partial_sort<NLC *>(NLC *, NLC *, NLC *)'
/include/stl_algo.h:1279:  instantiated from `__introsort_loop<NLC *, NLC, int>(NLC *, NLC *,
                                                                                NLC *, int)'
```

and continuing through functions in the bowels of the STL implementation that we did not even know existed. On the other hand, since `StaticIsA` lets us detect template constraint conformance at the instantiation point, we can use template specialization to dispatch "custom error messages" which succinctly report the problem at a level of abstraction clients will be able to understand (as we shall illustrate presently).

Consider again the `Sort()` function described above. We shall now use `StaticIsA` to generate a "custom error message" if the template type `T` does not conform to `LessThanComparable<T>`. The actual `Sort()` function does not do any work, it simply forwards the work to a helper class, which will be specialized on a boolean value:

```
template <class T> inline void Sort( vector<T>& v ) {
   SortHelper< StaticIsA<T,LessThanComparable<T> >::valid >::doIt( v );
}
```

The helper class is declared as

```
template <bool b> struct SortHelper;
```

and it has two very different specializations. The `true` version does the actual work as we would expect:

```
template <> struct SortHelper<true> {
   template <class T> static inline void doIt( vector<T>& v ) {
      // actually sort the vector, using operator< as the comparator
   }
};
```

The `false` version reports an error:

```
template <class T>
struct Error {};

template <> struct SortHelper<false> {
   template <class T> static inline void doIt( vector<T>& ) {
      Error<T>::Sort_only_works_on_LessThanComparables;
   }
};
```

Now, clients can do

```
vector<Foo> v; // recall: Foo isa LessThanComparable<Foo>
Sort(v);
```

and the vector gets sorted as we expect. But if clients try to sort a vector whose elements cannot be compared...

```
vector<NLC> v;
Sort(v);
```

...then our compiler says:

```
x.cc: In function `static void SortHelper<false>::doIt<NLC>(vector<NLC,allocator<NLC> > &)':
x.cc:73:   instantiated from `Sort<NLC>(vector<NLC,allocator<NLC> > &)'
x.cc:78:   instantiated from here
x.cc:67: `Sort_only_works_on_LessThanComparables' is not a member of type `Error<NLC>'
```

and nothing else.

The technique of trying to access

```
Error<T>::some_text_you_want_to_appear_in_an_error_message
```

seems to work well on different compilers; for all practical purposes, it lets one create “custom error messages”. Now the error is pinpointed, says what is wrong in no uncertain terms, and is not excessively verbose. This technique is mentioned in [CE00].

Custom error messages can be applied equally well to the algorithms in the STL by writing “wrappers” for STL functions. By supplying a wrapper for functions like `std::sort()`, we can create a function that is as efficient as `std::sort` (it simply forwards the work with an inline function that an optimizing compiler will easily elide), but will report meaningful errors if the type of element being sorted is not a `LessThanComparable`. Rather than getting fifty-seven lines of gobbledegook from an improper instantiation of `std::sort()`, we can get four lines of meaningful messages from our own version of `sort()`.

The astute reader may be wondering what will happen with code like this:

```
vector<int> v;
Sort(v);
```

Clearly `int` does not (and cannot) inherit from `LessThanComparable`. However, we would like this to compile successfully. We use the usual "traits" trick to solve this problem; each static interface has an associated traits class, which can be specialized for types which conform to the interface "outside of the framework". Put another way, traits provide a way to declare conformance extrinsically.

We mentioned the `MAKE_TRAITS` macro before; here is its definition:

```
#define MAKE_TRAITS \
   template <class Self> \
   struct Traits { \
      static const bool valid = false; \
   };
```

For any static interface `SI`, `SI::Traits<T>::valid` says whether or not `T` "isa" `SI` "outside the framework" (that is, `T` exhibits named conformance to `SI` despite the lack of an inheritance relationship). To declare named conformance extrinsically, we just specialize the template. For example, to say that `int` isa `LessThanComparable<int>`, we say

```
template <>
struct LessThanComparable<int>::Traits<int> : public Valid {};
```

where `Valid` is defined as just

```
struct Valid {
   static const bool valid = true;
};
```

More generally

```
template <>
struct SomeStaticInterface::Traits<SomeType> : public Valid {};
```

declares that `SomeType` "isa" `SomeStaticInterface`.

At this point, we should take a moment to describe the behavior of

```
StaticIsA<T,SI>::valid
```

which comprises the "brains" of our static interface approach. The behavior is essentially the following:

```
if "T isa SI" according to the "traits" of SI, then
   return true
else if T inherits SI (named conformance), then
   apply the structural conformance check
   return true
else
   return false
```

Note that the structural conformance check will issue a compiler-error if structural conformance is not met. For example, if class `Foo` inherits `LessThanComparable<Foo>`, but does *not* define an `operator<`, then this structural conformance error will be diagnosed when

```
StaticIsA<Foo,LessThanComparable<Foo> >::valid
```

is first evaluated (and thus the `check_structural()` member of the static interface is instantiated). The error message will be generated by the compiler (not a "custom error message" as described above), as there is no general way to detect the non-existence of such members before the compiler does. This is unfortunate, as such errors tend to be verbose. Fortunately, these kinds of errors are errors by suppliers, not clients. As a result, they only need to be fixed once.

2.2 Using static interfaces for static dispatch

Just as we can use template specialization to create custom error messages, we can similarly use it to dispatch the appropriate implementation of a template function or class, based on the concepts modeled by the template argument. For example, one could implement a `Set` with a hashtable, a binary search tree, or a linked list, depending on if the elements were `Hashables`, `LessThanComparables`, or `EqualityComparables` respectively. The code below says exactly that: clients may use `Set<T>`, and `Set` will automatically choose the most effective implementation (or report a custom error message if no implementation is appropriate):

```
enum { SET_HASH, SET_BST, SET_LL, SET_NONE };
template <class T> struct SetDispatch {
   static const bool Hash = StaticIsA<T,Hashable>::valid;
   static const bool LtC  = StaticIsA<T,LessThanComparable<T> >::valid;
   static const bool EqT  = StaticIsA<T,EqualityComparable<T> >::valid;
   static const int which = Hash?SET_HASH: LtC?SET_BST: EqT?SET_LL: SET_NONE;
};
template <class T, int which = SetDispatch<T>::which > struct Set;
template <class T> struct Set<T,SET_NONE> {
   static const int x = Error<T>::
Set_only_works_on_Hashables_or_LessThanComparables_or_EqualityComparables;
};
template <class T> struct Set<T,SET_LL> {
   Set() { cout << "Set list" << endl; }
};
template <class T> struct Set<T,SET_BST> {
   Set() { cout << "Set bst" << endl; }
};
template <class T> struct Set<T,SET_HASH> {
   Set() { cout << "Set hash" << endl; }
};
```

[Side note: `SetDispatch` might best be implemented with `enum`s rather than `static const` variables, but we encountered bugs in our compiler when using `enum`s as template parameters.] The code above is simple: `SetDispatch<T>::which` computes information about the conformance of `T` to various static interfaces; `Set` then uses this information to dispatch the appropriate implementation (or a custom error message).

3. The Design Space of Constraint Checking

In the previous sections, we have described the most “radical” features of our framework. We chose to present these features first, as they are the key features that set our framework apart from other concept-checking ideas that we are aware of. In fact, however, our framework gives the user a choice among concept-checking approaches which span the concept-checking design space. We describe the design space and the components of our system in this section.

3.1 Traditional concept-checking approaches

Stroustrup describes a simple constraints mechanism in [Str94]. An up-to-date generalization of that approach can be seen in this code:

```
   template <class T>
   struct TraditionalLessThanComparable {
      T y;
      template <class Self>
      void constraints(Self x) {
         bool b = x<y;
         (void) b;        // suppress spurious warning
      }
   };

   template <class T, class Concept>
   inline void Require() {
      (void) ((void (Concept::*)(T)) &Concept::template constraints<T>);
   }

   // just inside any template that wants to ensure the requirement:
   Require< Foo, TraditionalLessThanComparable<Foo> >();
```

A very similar approach is used in the latest versions of the SGI implementation of the STL [SGI]. We will call this approach *traditional concepts* and contrast it with static interfaces. Note that the benefit of traditional concepts is simply that the compiler will first issue its error messages at the template instantiation point, rather than deeper in the bowels of the implementation. The key differences between traditional concepts and static interfaces are:

1. Traditional concepts use no named conformance; they are entirely structural.

2. Traditional concepts *call* methods in the required interface (`x<y`) rather than taking pointers to them.

The consequence of the first difference is that client classes (e.g., models of `TraditionalLessThanComparable`) do not need to explicitly state that they conform to the `TraditionalLessThanComparable` protocol. In practical terms, this is a good property for legacy code and third-party libraries (including the STL) but a dangerous property for new development, because of the possibilities of accidental conformance. Its philosophy is also contrary to the design model of the C++ language that uses named inheritance (a subtype explicitly specifies its supertype) instead of structural inheritance.

The consequence of the second difference (calling methods instead of taking pointers) is that traditional concepts are more “forgiving” than static interfaces. Indeed, static interfaces are strict in two ways. First, they require exact type conformance for the method signatures. For instance, if a method is expected to accept two integer arguments, static interfaces will reject a method accepting arguments of a type to which integers implicitly convert. Second, taking a pointer to a member of a type `T` does not allow us to check for non-member functions that take a `T` as an argument (most commonly, overloaded operators), as the language does not allow us to unambiguously make a pointer to such non-members functions without knowing the namespace these functions are defined in. This is perhaps a defect in the language standard. Koenig lookup (section 3.4.2 of [ISO98]) allows us to *call* these same functions that we cannot take pointers to. Nevertheless, Koenig lookup does not apply in the context of overload resolution when taking pointers to non-member functions (that is, in 13.4 of [ISO98]). Note that the latter is a limitation of static interfaces (i.e., the scheme is being unnecessarily strict beyond our control).

Again, we see that the trade-off is quite similar to before. For legacy or third party code, one may want to be as “forgiving” as possible, and, thus, the “traditional” concept checking described above makes sense. For a single, controlled project, however, static interfaces are more appropriate, as they allow expressing strict requirements. Instead, traditional concepts make a “best effort” attempt to catch some common errors, but do not provide any real guarantees of doing so.

For an illustration consider the example of `TraditionalLessThanComparable`, shown earlier. Writing out expressions (like “`x<y`”) to check concepts is particularly error-prone, by virtue of the many implicit

conversions available in C++. The problem is more evident in the return types of expressions. A class satisfying the `LessThanComparable` concept should implement a “<” operator that is applied on given types. Nevertheless, it is hard to ensure that the return type of this operator is exactly `bool` and not some type that is implicitly convertible to `bool`. The latter can cause problems. As an example, consider this code:

```
template <class T>
void some_func() {
   Require< T, TraditionalLessThanComparable<T> >();

   T a, b, c, d;
   if( a<b || c<d )  // Require() should ensure this is legal
      std::cout << "whatever" << std::endl;
}
```

We would expect that any problems with the code will be diagnosed when `Require()` is instantiated. Nevertheless, it is possible to get past `Require()` without the compiler complaining, but then die in the body of the function. Here is one such scenario:

```
struct Useless {};

struct EvilProxy {
   operator bool() const { return true; }
   Useless operator||( const EvilProxy& ) { return Useless(); }
};

struct Bar {
   EvilProxy operator<( const Bar& ) const { return EvilProxy(); }
};

some_func<Bar>();
```

Indeed, we have been foiled by implicit conversions. `Bar`'s `operator<` does not, in fact, return a `bool`; it returns a proxy object (implicitly convertible to `bool`). This proxy is malicious (as suggested by its name) and conspires to make the expression

```
a<b || c<d
```

of type `Useless` rather than type `bool`.

Hopefully, template authors do not often run into such contrivedly “evil” instantiations. Nevertheless, the point remains that implicit conversions can lead to surprises. The SGI STL implementation is susceptible to problems of this kind. There are several examples one can devise that evade the concept checking mechanism, although they are not legal instances of the documented STL concepts.

The case of static interfaces vs. traditional concepts is an instance of the classic design trade-off of detection mechanisms: one approach avoids most false-positives at the expense of producing many false-negatives, while the other does just the reverse. Ideally we would like the minimize both false-positives (constraints that admit code which may turn out to be illegal or not meaningful) and false-negatives (constraints that reject code which should be acceptable).

3.2 Enhancing Traditional Concepts

It is possible to strengthen traditional concepts in order to make them less susceptible to implicit conversion attacks. Recall our example of a traditional concept with a `constraints()` function with

```
bool b = x<y;
```

as the check. This check is vulnerable because implicit conversion can (purposely or inadvertently) make this code legal, without satisfying the implicit requirements. The best solution that we have found involves our own kind of proxy object. This object is designed to protect us from the “evil proxies” in the world, and hence we call it “HeroicProxy”. We show the code for it and how it is used here:

```
template <class T>
struct HeroicProxy {
   HeroicProxy( const T& ) {}
};

template <class T>
struct MaybeOptimalLessThanComparable {
   const T y;
   template <class Self>
   void constraints(const Self x) {
      HeroicProxy<bool> b = x<y;
      (void) b;        // suppress spurious warning
   }
};
```

`HeroicProxy` takes useful advantage of the rules for implicit conversion sequences for binding references ([ISO98] sections 13.3.3.1.4, 8.5.3 and others). We are assured that the result of `x<y` is not some user-defined proxy object; the result might not be a `bool`, but at worst it will be another “harmless” type like `int`. We have also `const`-qualified `x` and `y`, to ensure that `operator<` isn't trying to mutate its arguments. `MaybeOptimalLessThanCompare` and the `HeroicProxy` comprise a middle-of-the-road approach, which seeks to minimize both the false-positives and false-negatives of the structural constraint detection.

3.3 A Hybrid Approach

In the previous sections, we have shed light on the continuum of structural checking that “template constraints” may enforce. Although our primary focus and novelty of our work is on static interfaces, *we also supply enhanced traditional concepts in our framework*. Users of our framework are, of course, encouraged to utilize the methods that are most suitable to their particular domain.

We now explain all the components of our framework. Given a static interface, which takes the general form

```
struct AParticularStaticInterface {
   MAKE_TRAITS;
   template <class Self>
   void check_structural( Self ) {
      // checking code, using any of the strategies just described
   }
protected:
   ~AParticularStaticInterface() {}
};
```

we define the following five “components” that users can use to specify template constraints.

```
(1) StaticIsA< T, AParticularStaticInterface >::valid
```

This checks both named and structural conformance using static interfaces, and returns the named conformance as a compile-time constant boolean.

```
(2) Named< T, AParticularStaticInterface >::valid
```

This checks only named conformance, resulting in a compile-time constant boolean.

```
    (3) RequireStructural< T, AParticularStaticInterface >()
```

This applies the structural check without the named check by using traditional concepts.

```
    (4) RequireNamed< T, AParticularStaticInterface >()
```

This enforces the named check; equivalent to generating a custom error iff `Named<...>::valid` is `false`.

```
    (5) RequireBoth< T, AParticularStaticInterface >()
```

This enforces the `StaticIsA` check; equivalent to generating a custom error iff `StaticIsA<...>::valid` is `false`.

4. Limitations and Extensions

Detecting structural conformance. The first limitation is one that we have already mentioned: we cannot *detect* structural conformance, rather we can only *ensure* it with a compile-time check. C++ apparently does not have a general way to detect the presence of members in a template type. Nevertheless, as mentioned above, this is only a tiny hindrance: if there is an error with a supplied class which purports to conform to a static interface, the compiler will emit its own error message (in the instantiation of the `check_structural()` method in a static interface), and the problem can be fixed once-and-for-all in that class (it is not a problem for clients).

Also, since our "custom error messages" and "static dispatch" (both described in Section 2) both require that we can *detect* conformance, this means they can only be used with *named* conformance techniques.

Other kinds of constraints. Finally, we have only demonstrated constraints based on normal member functions. One can imagine constraints based on other properties of a type, such as public member variables, member template functions, nested types, etc. There are various clever tricks one can implement in `check_structural()` to ensure some of these types of members. To unobtrusively ensure the existence of a public member variable, simply try to take a pointer to it. To ensure that a nested type exists, try to create a `typedef` for it. To ensure that a method is a template method, declare an empty private `struct` called `UniqueType` as a member of the static interface class, and call the method with a `UniqueType` as a parameter.

5. Related work

In [Str97], exercise 13.16, Stroustrup suggests constraining a template parameter. This exercise motivated our work; our original (simpler) solutions were poor (they either had no structural conformance guarantees or introduced needless overhead with `virtual`) and we sought to improve upon them.

The idea of constraining template parameters has been around for a while--at least since Eiffel [Mey97]. Some languages, like Theta [DGLM95], have mechanisms to explicitly check structural conformance at the instantiation point, but lack named conformance. This allows for accidental conformance. Recent languages like GJ [BOSW98] support both named and structural conformance checks--including F-bounded constraints, which are necessary to express the generalized form of many common concepts (like `LessThanComparable`). Clearly our work is intended to provide the same kind of mechanism for C++, though our implementation goes beyond this, allowing code to use static interfaces for more than just constraints (e.g., we can do selective implementation dispatch based on membership in a static interface hierarchy).

Alexandrescu [Ale00a] devises a way to make "traits" apply to an entire inheritance hierarchy (rather than just a single type). This seems to enable the same functionality as our selective implementation dispatch, only without structural conformance checks.

Alexandrescu also presents an "inheritance detector" in [Ale00b]. Our framework employs this detector as a general purpose named-conformance detection-engine, which is at the heart of our system.

Baumgartner and Russo [BR95] implement a signature system for C++, which is supported as an option in the g++ compiler. Unlike our work, the emphasis of this system is on structural subtyping, so that classes can conform to a signature without explicitly declaring conformance. In practice, the system is only usable in the context of runtime polymorphism for retrofitting an abstract superclass on top of pre-compiled class hierarchies. Thus, the Baumgartner and Russo work is an alternative implementation of *dynamic*, and not static, interfaces.

6. Evaluation and Conclusions

Static interfaces allow us to express "concepts" in C++ and have them enforced by the compiler. Our `StaticIsA` mechanism is a reusable way to detect and ensure named and structural conformance. Users of our framework can define their own interfaces, as well as types that conform to those interfaces, and then use `StaticIsA` to ensure that templates are instantiated properly.

We have demonstrated idioms that rely on `StaticIsA` to create informative error messages which are a significant improvement over those error messages the compiler gives. This is a result of being able to directly express "concepts" inside C++. Our techniques also make code more self-documenting.

We have witnessed much "ad hoc trickery" to mimic template constraints in the past. We believe the techniques described here provide a more effective, general, and reusable strategy than previous attempts, and we hope our work will help evolve C++ to better meet the demands of future template programmers.

Our source code can be found at:

`http://www.cc.gatech.edu/~yannis/static-interfaces/`

7. References

[Ale00a] Alexandrescu, A. "Traits on Steroids." C++ Report 12(6), June 2000.

[Ale00b] Alexandrescu, A. "Generic Programming: Mappings between Types and Values." C/C++ Users Journal, October 2000.

[BR95] Baumgartner, G. and Russo, V. "Signatures: A language extension for improving type abstraction and subtype polymorphism in C++." Software Practice & Experience, 25(8), pp. 863-889, Aug. 1995.

[BOSW98] Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P. "Making the future safe for the past: Adding Genericity to the Java Programming Language." OOPSLA, 1998.

[CE00] Czarnecki, K. and Eisenecker, U. Generative Programming. Addison-Wesley, 2000.

[DGLM95] Day, M., Gruber, R., Liskov, B. and Myers, A. "Subtypes vs. Where Clauses: Constraining Parametric Polymorphism." OOPSLA, 1995.

[ISO98] ISO/IEC 14882: Programming Languages -- C++. ANSI, 1998.

[Mey97] Meyer, B. Object-Oriented Software Construction (Second Edition). Prentice Hall, 1997.

[SGI] Standard Template Library Programmer's Guide (SGI). `http://www.sgi.com/Technology/STL/`

[Str94] Stroustrup, B. The Design and Evolution of C++. Addison-Wesley, 1994.

[Str97] Stroustrup, B. The C++ Programming Language (Third Edition). Addison-Wesley, 1997.