Identification of Idiom Usage in C++ Generic Libraries

Andrew Sutton, Ryan Holeman, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio 44242
{asutton, rholeman, jmaletic}@cs.kent.edu

***Abstract*—A tool supporting the automatic identification of programming idioms specific to the construction of C++ generic libraries is presented. The goal is to assist developers in understanding the complex syntactic elements of these libraries. Large C++ generic libraries are notorious for being extremely difficult to comprehend due to their use of advanced language features and idiomatic nature. To facilitate automated identification, the idioms are equated to micropatterns, which can be evaluated by a fact extractor. These micropattern instances act as beacons for the idioms being identified. The method is applied to study a number of widely used open source C++ generic libraries.**

Keywords—C++, Templates, Generic Libraries, Emiprical Study

I. INTRODUCTION

Generic programming is a paradigm that equips developers with a mechanism for designing adaptable and efficient generic algorithms and data structures. In C++, the paradigm is rooted in the ability to parameterize (via templates) algorithms and data structures with user-specific types at compile time in order to compose more complex software abstractions. While extensive use of the compiler may yield substantial benefits in terms runtime performance and elegant design, the resulting software is typically very difficult to comprehend due to the idiomatic nature of generic programming in C++. Frequently composed of cryptic template definitions and instantiations, generic libraries can often perplex even the most seasoned software engineer.

The incomprehensibility of C++ generic libraries can be largely attributed to the lack of supporting linguistic abstractions, which results in the abuse of existing language structures (e.g., classes) to develop programming idioms in their place. Whereas languages that embrace other paradigms (esp., object-oriented programming) have evolved features that more readily express their idioms and abstractions, the language features underlying generic programming in C++ exists at the most basic level: class and function templates.

In order for a programmer to become proficient in the generic paradigm, they must master the idioms and patterns that define its abstractions, which in the case of C++ implies a practical mastery of the entire language and the idiosyncrasies of its compilers. On the other hand, one purpose of generic libraries is to provide reusable and adaptable libraries for application developers. Given that generic libraries (via templates) are necessarily *transparent* (i.e., not black boxes), their usage creates a *leaky abstraction*. The idioms used in the construction of generic libraries are leaked to more casual developers in the form of error messages and, in some cases, compilation failures.

Our goal is to assist developers in understanding patterns and idioms in generic libraries by providing tools to automatically identify them in source code. This helps address the "leakiness" of advanced generic and generative programming technique by mapping these implementation-level idioms into more comprehensible, abstract forms. In this work, we survey a number of prevalent programming idioms used in widely-used C++ generic libraries. By casting idioms as micropatterns, we describe a mechanism by which their instances can be automatically identified in source code. We present and validate a tool that automatically identifies instances of these micropatterns in C++ source code and use this tool to study a number of open source generic libraries. Results of the study are used to motivate discussion on tools and techniques for using and/or maintaining generic libraries and aspects of the evolution of the C++ programming language along the same lines.

This paper is organized as follows. Section 2 presents related work. In section 3, we describe commonly used idioms in generic programming, and in section 4 we decompose those idioms into micropatterns. Section 5 presents the implementation and its validation. In section 6 we describe the study of a number of widely used C++ generic libraries, and present discussion and conclusions in Section 7 & 8.

II. RELATED WORK

The style of generic programming that dominates the C++ landscape was pioneered in Ada [16] and later adapted to C++ templates [17] where it formed the basis of the Standard Template Library (and later a part of the C++ Standard Library) [3]. This work includes techniques for functors, tag dispatch, and traits classes. The Boost Graph Library, for example, uses these techniques extensively to implement generic graph algorithms [19].

A number of techniques related to this style of programming have been discussed in the literature. Coplien provides the first description of common template patterns in generic components [5]. Specific design patterns for generic programming are presented in [8], and generative techniques for datatype composition (i.e., mixins) are described in [22]. Template metaprogramming and its associated idioms/patterns are described by Abrahams and Gurtovoy [1]. Alexandrescu employs a number of these techniques in his treatment of C++ library design [2].

Despite the wealth of work on techniques for generic programming and library design, there is comparatively little work on program analysis or reverse engineering of generic libraries or templates. There are a few notable exceptions. A *lint*-like tool is presented in [12] that diagnoses misuses of generic concepts (especially iterators) in the STL. A method of measuring the impact of changes to concept definitions is presented in [26]. An approach to support the debugging of template metaprograms is given in [18]. A method of recovering type constraints (concepts) from function templates is presented in [24], and a technique to support program analyses of template instantiations is given in [23].

This work is an extension of the study conducted in [13] on identifying programming idioms in C++ generic libraries. We have extended this work by adapting it to the notion of micropatterns [10], which are described as class-level patterns that can be expressed as a "simple formal condition on the attributes, types, name, and body". Many of the programming idioms identified in the previous study can be expressed as predicates on classes in generic libraries. We have also expanded the set of idioms being identified.

III. IDIOMS FOR GENERIC PROGRAMMING

In this section, we present a number of common programming idioms used in C++ generic library design. We do not claim to describe all known idioms for generic programming in C++, only those we perceive as the most frequently used. In the presentation of these idioms, we sometimes omit the `typename` specifier, for brevity.

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

Figure 1. The `less` functor encapsulates the comparison, via `operator<`, for a type `T`.

A. Functors

A functor (sometimes called a *function object*) is a class that overloads the function call operator (`operator()`). Functors are used to support the definition of high-order functions in C++. By implementing a function as an object, functors be used to a) abstract or adapt calls to other functions, b) curry function parameters, c) maintain state (including external data references) between calls. An example of a functor is given in Figure 1.

Here, the class template `less` is a functor that abstracts the syntactic comparison of objects via `operator<`. This can be used to parameterize high order functions (algorithms) such as `sort`.

B. Template Metaprogramming

Template metaprogramming encompasses the definition and use of programs that perform compile-time operations on types or integral constants, and is predicated upon the definition of template metafunctions [1]. A template metafunction is class (usually a template) that "returns" a computed result as a typedef or static constant. A non-template metafunction is called a *constant metafunction*; the Boost.MPL types `true_` and `false_` can be thought of as constant metafunctions. A metafunction that returns a type (via a typedef) is called a *type metafunction*, and a metafunction that returns a value (via a static constant or enumeration value) is called an *integral metafunction*, and if the integral type is `bool`, then we refer to it as a *boolean metafunction*. We use the term "return" loosely. In reality, the "caller" of a metafunction accesses the "return" value by referring to a nested declaration.

There are generally two uses of metafunctions: the definition of integral metaprograms (e.g., GCD and LCM) and the manipulation or selection of types in generic libraries. The latter application is of more interest in this work. Metafunctions that operate on types are frequently used to select and refine class structure, enhance C++ overloading, and act as aliases to more complex type definitions. For example, consider the type metafunction (also called a *type accessor*) shown in Figure 2.

```
template<typename T, typename U>
struct add_result {
  typedef decltype(T() + U()) type;
};
```

Figure 2. The `add_result` type accessor yields the result type of the `operator+` defined for the `T` and `U`.

The `add_result` type accessor demonstrates an interesting pattern in C++, namely the use of metafunctions functions to deduce the result type of an expression.

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

Figure 3. The `is_same` type trait is comprised of a class template that responds in the negative and a partial specialization that responds in the affirmative iff the template is instantiated over the same types.

Another common use of metafunctions is that of type traits. *Type traits* are typically classified as any metafunction query or transformation on a type (or types) with respect to some domain. Many of these queries are Boolean metafunctions, and are implemented as a set of class template specializations. The standard header

<type_traits> defines a number of traits that can be used to evaluate common properties of types in the C++ type system. For example, consider an implementation of the is_same type trait shown in Figure 3.

Another used with metafunctions is the definition of metafunction classes. A metafunction class is a class (frequently a template) that defines a nested metafunction (generally named "apply"). Metafunction classes enable high-order metafunction programming by allowing lazy (or deferred) instantiation of the metafunctions. This technique is frequently used to wrap template names, projecting them into the type system. For example, the Boost Graph Library (BGL) uses a variant of this approach to select and instantiate containers for vertex and edge lists. The vector selector is shown in Figure 4.

```
struct vecS {
  template<typename T>
  struct bind_ {
    typedef std::vector<T> type;
  };
};
```

Figure 4. The vecS metafunction class uses an inner bind_ operator to the requested container.

The technique used here allows the adjacency list implementation to select features when the container is known to be vecS, but before the underlying vector type has been instantiated. In this particular case, instantiation of the vector is delayed because of cyclic dependencies between types in the definition of the graph class. We note that this definition is only used when the compiler does not correctly support partial template specialization.

C. Tag Dispatch

Tag dispatch is a technique that is used to enhance function overloading on different *kinds* or categories of types. This technique is used when different algorithms can be identified for similar but related abstractions within a generic domain. The tag dispatch idiom relies on the definition of tag classes and the overloading of functions on those types.

Specifically, there are four components to the tag dispatch idiom: a set of tag classes, a map function, the dispatcher, and the dispatch targets. A *tag class* is an empty class that describes a kind of type or a property of a type. Tag classes are sometimes organized inheritance hierarchies (called tag class hierarchies) in order to categorize types. The *map function* is a function or metafunction that maps a type to its most specialized tag class.

The dispatch framework is comprised of a dispatcher and set of targets. The *dispatch targets* have mostly the same parameters but are overloaded on different tag classes. The dispatcher invokes a target calling the overloaded name with the expected arguments and the result of the map function. The compiler will statically resolve the overload, and the use of inlining and copy elision will generally guarantee that the dispatch technique incurs no runtime overhead.

One of the best known examples of tag dispatch is the STL advance operation for iterators. The entire series of structures is given in Figures 5-7.

```
struct in_iter_tag { };
struct fwd_iter_tag : in_iter_tag { };
struct bidi_iter_tag : fwd_iter_tag { };
struct ra_iter_tag : bidi_iter_tag { };
```

Figure 5. The STL iterator tag class hierarchy classifies iterators based on their traversal operations and properties.

```
template<typename Iter>
typename Iter::category
iterator_category(Iter) {
  return Iter::category();
}
template<typename T>
ra_iter_tag iterator_category(T*) {
  return ra_iter_tag();
}
```

Figure 6. The iterator_category function maps an iterator type to its corresponding tag class. An overload is required for pointer types.

```
template<typename Iter>
void adv(Iter& i, int n, in_iter_tag) {
  assert(n >= 0);
  while(n--) ++i;
}
template<typename Iter>
void adv(Iter& i, int n, bidi_iter_tag) {
  if(n > 0) while(n--) ++i;
  else      while(n++) --i;
}
template<typename Iter>
void adv(Iter& i, int n, ra_iter_tag) {
  i += n;
}
template<typename Iter>
void advance(Iter& i, int n) {
  advance(i, n, iterator_category(i));
}
```

Figure 7. The advance algorithms are dispatched for different kinds of iterators based on the iterator tag hierarchy. The last implementation is the dispatcher.

The STL defines a tag hierarchy for iterator, which is shown in Figure 5. The tag class hierarchy defines four distinct classifications of iterators based on their traversal properties—how an iterator moves along or consumes a sequence of objects. Each kind of iterator is represented by a tag class in this hierarchy. An input iterator allows forward traversal through a sequence. A forward iterator is a multi-pass input iterator (the sequence is not consumed during traversal). A bidirectional iterator can also traverse backwards in the sequence, and a random access iterator can move multiple steps in either direction in constant time. Iterator types are mapped to the iterator hierarchy via the iterator_category function, whose implementation is shown in Figure 6

The advance algorithm moves an iterator by *n* steps. The values of *n* allowed by the algorithm and the efficiency with which the move is computed vary with the kind of iterator.

If the iterator is a forward iterator (e.g., singly linked lists), *n* must be positive, but if the iterator is bidirectional *n* may also be negative. The algorithm executes in *O(1)* if and only if the iterator supports random access, *O(n)* otherwise. The advance algorithm and its dispatch targets are shown in Figure 7.

D. Traits Class

A *traits class* is a class template that adapts a type to a generic abstraction (or concept) by providing associated types and static functions and (constant) data. A traits class decouples the access of these elements from actual implementation types allowing generic algorithms and data structures to be written in terms of the abstraction rather than the implementations. This is especially useful if the implementations provide varied interfaces to similar functionality. A traits class typically has two components: a generic definition and a set of specializations that adapt types to that definition.

The canonical example of traits classes is the STL's `iterator_traits`. The `iterator_traits` class makes it possible to write algorithms in terms of a generic iterator abstraction by providing a mechanism for adapting arbitrary types to the iterator concept. The sole purpose of this adaptive technique is to decouple the access of associated types from the iterator. This is required for pointer types, which do not define nested types (e.g., `value_type`, `reference`, `pointer`, etc.).

```
template<typename Iter>
struct iterator_traits {
  typedef Iter::value_type value_type;
  typedef Iter::reference reference;
  typedef Iter::pointer pointer;
};
template<typename T>
struct iterator_traits<T*> {
  typedef T value_type;
  typedef T& reference;
  typedef T* pointer;
};
```

Figure 8. The `iterator_traits` class derives a number of different associated types for its `Iter` template parameter. A partial specialization adapts pointer type to the generic definition.

The `iterator_traits` class acts as a kind of parameterized façade for a number of different type accessors on its `Iter` template parameter. In some cases, however, it is not possible for an iterator implementation to define these associated types. For example, instantiating this template over pointer types (i.e., `T*`) will lead to compiler errors—even though `T*` is a valid (random access) iterator—because `T*` cannot have any associated types. This enables generic algorithms to be written in terms traits classes rather than specific types.

The generic `iterator_traits` class implements an adaptor for conformant types. Any iterator implementation can be made to conform to the iterator concept by defining the correct associated types. This is shown in Figure 8.

E. Constrained Templates

There are cases, in the design of generic data structures or algorithms, where it becomes useful to enable or disable a subset of specializations or overloads based on the properties of the types over which they are being instantiated. *Concept-controlled polymorphism* [14, 15] is similar to tag dispatch except that template instantiations are *enabled* or *disabled* by a metafunction rather than being *selected* by a tag class.

This technique of enabling or disabling template instantiation is rooted in the use of SFINAE (Substitution Failure Is Not An Error) to quietly remove specializations or overloads from the list of templates being considered for instantiation and overload resolution. Although SFINAE can be applied casually in a large number of situations, its systematic use is typically predicated upon the use of SFINAE-based *enablers* and *disablers*. SFINAE enablers are designed to trigger substitution failures based on the evaluation of Boolean metafunctions and are, themselves, typically implemented as a kind of "partial" metafunction. A simplified version the Boost C++ Library's SFINAE enabler, called `enable_if`, is shown in Figure 9.

```
template<typename C, typename T = void>
struct enable_if {
  typedef T type;
};
template<typename T>
struct enable_if<false_, T> { };
```

Figure 9. The `enable_if` template implements a SFINAE enabler that can be used to control the instantiation of templates based the evaluation of a metafunction `C`.

We say that `enable_if` is a partial metafunction since only one specialization defines the requisite associated type. If the template is instantiated in such a way that C is substituted with the constant metafunction `false_`, then the associated type will not be defined in the resulting template instance, and a subsequent reference to `type` will generate a substitution failure. A SFINAE *disabler* simply inverts the logic of the enabler.

SFINAE enablers are used as either the result type of a function declaration or as the type of an additional function parameter that defaults to `nullptr`. Consider the operation `infinity` shown in Figure 10, in which the constraint is applied to the result type.

```
template<typename T>
enable_if<is_floating_point<T>, T>::type
infinity() {
  return numeric_limits<T>::inf();
}
template<typename T>
disable_if<is_floating_point<T>, T>::type
infinity() {
  return numeric_limits<T>::max();
}
```

Figure 10. The first overload of `infinity` is enabled if and only if `T` satisfies the `is_floating_point` predicate.

If `is_floating_point<T>` evaluates to `true_` (e.g., called as `infinity<double>()`) then the first overload is instantiated correctly and the second overload is quietly removed from the candidate set. Conversely, if the template argument is given as `int`, then the first overload will be excluded and the second selected.

F. Partial Templates

Another technique used to selectively allow (or disallow) template instantiation is to declare (but not define) a class or function template and then only provide specializations or overloads for the intended targets. The intent of the *partial template* idiom is to explicitly restrict the set of valid arguments over which it can be instantiated. For example, consider the `container_gen` facility in the BGL, which is used to instantiate vertex and edge containers based on a selectable tag class. Its implementation is shown in Figure 11.

```
template<typename Tag, typename Value>
struct container_gen { };
template<typename Value>
struct container_gen<vecS, Value> {
  typedef std::vector<Value> type;
};
template<typename Value>
struct container_gen<listS, Value> {
  typedef std::list<Value> type;
};
```

Figure 11. The partial template `container_gen` must be parameterized over a valid container selector (`Tag`) class and value type.

If the instantiation of `container_gen` includes a tag class that is not specialized on, this will almost certainly result in a compiler error. Another common technique is to declare, but not define the primary template. This is common with recursive, variadic templates in C++0x.

G. Mixins

A *mixin* is a class that can be used to "inject" functionality into a user-defined type [22, 25]. In C++, mixins typically take the form of a class template that derives from one of its template parameters. This technique allows programmers to construct or compose data structures that aggregate the functionality of their mixins.

```
template<typename Base = null_arch>
struct def_ctor_arch : Base {
  def_ctor_archetype();
};
template<typename Base = null_archetype>
struct copy_ctor_arch : Base {
  copy_ctor_arch(copy_ctor_arch const&);
};
```

Figure 12. The default constructible archetype uses the mixins pattern to support interface aggregation.

For example, mixins are used in the archetype system of the Boost C++ libraries to support concept-checking tests. An archetype is a class that exposes only the structural requirements of a concept (or template type constraint), no more, no less. They are used to evaluate the specification of concepts in generic algorithms and data structures. Figure 12 shows an abbreviated version of the Boost archetype that enables tests for default and copy constructability.

Here, two archetype classes expose functionality for constructing types that are either copy constructible or default constructible. A new archetype that is both copy and default constructible can be composed as the type `copy_ctor_arch<def_ctor_arch<>>`. The resulting class should expose both constructors, but no more.

H. Curiously Recurring Template Pattern

The *Curiously Recurring Template Pattern* (CRTP) is an inheritance pattern that most frequently used to provide default implementations of common operations for a user-defined type [5]. In this idiom, the base class provides services that depend on operations in the deriving class. In a sense, this is analogous to dynamic polymorphism where the base class invokes virtual or abstract methods that are intended to be overridden or implemented by the derived class. Instead of virtual methods however, this base class statically casts itself as the derived class and invokes the needed method. CRTP is largely a utility mechanism that is frequently used to simplify the process of adapting types to a known interface.

```
template<typename Derived, typename Value>
class iterator_facde {
  typedef Value value_type;
  typedef Value& reference;
  typedef Value* pointer;
  Derived* self() {
    return static_cast<Derived*>(this);
  }
  Derived& operator++() {
    self()->increment();
    return *self();
  }
  reference operator*() {
    return self()->dereference();
  }
  pointer operator->() {
    return &(self()->dereference());
  }
};
```

Figure 13. A partial implementation of Boost's `iterator_facade` is parameterized over a `Derived` iterator implementation and its `Value` type. Standard iterator operations are implemented in terms of functions defined by the `Derived` type.

For example, consider a partial implementation of the Boost Iterator Library's `iterator_facade` class. The `iterator_facade` is a class template that is parameterized over a user-defined class that can be adapted to act as an iterator. The user-defined class is required to supply several methods to work with the base.

The `iterator_facade` class is parameterized over two types, the first of which is a user-defined class that implements some form of iterator functions (this is the CRTP parameter), and the second is the value type of the iterator, which is used to determine the reference type. One of the

TABLE 1. THE MICROPATTERN CLASSIFICATIONS AND DESCRIPTIONS ARE EXTENDED AND ADAPTED TO ADDRESS C++ AND GENERIC PROGRAMMING. THE IDIOM REPRESENTED BY THE MICROPATTERN, IF ANY, IS GIVEN IN THE 3RD COLUMN

Degenerate Classes

Degenerate State and Behavior		
Designator	A non-template class with no members	Tag Class
Taxonomy	A Designator that derives from one other class	Tag Class
Joiner	A Designator that derives from multiple classes	Tag Class
Traits Class	A class template with only typedefs and possibly static (constant) members	Traits Class
Metaprogramming		
Type Metafunction	A class declaring an associated type named 'type'	Metafunction
Integral Metafunction	A class declaring a static constant integral attribute named 'value'	Metafunction
Constant Metafunction	A metafunction that is not a class template	Metafunction
Metafunction Class	A class declaring a nested Metafunction named 'apply'	Metafunction
Degenerate Behavior		
Functor	A class that overloads the function call operator	Functor
Function Class	A class with a single static method and no attributes	
Degenerate State		
Stateless	A class with no attributes.	

Containment

Data Managers		
Record	A class with only non-reference, public attributes	
Environment	A class whose attributes are all references	

Inheritance

Base Classes		
Static Outline	A class that statically down-casts 'this' as a dependent type	Curious Template
Inheritors		
Mixin	A class that derives from a template parameter	Mixin

Specialization

Degenerate Specialization		
Enabler	A Metafunction with a specialization that is not a Metafunction	Template Constraint
Partial Template	An undefined or empty class template with non-empty specializations	Partial Template

most important feature of this class is the `self` function, which statically downcasts `this` object to its `Derived` class. The required iterator interface, the operators `++`, `*`, and `->` are provided by the façade. The class represented by the type parameter `Derived` must implement the functions `increment`, `decrement`, and `dereference`.

IV. MICROPATTERNS IN GENERIC LIBRARIES

In order to study the use of these idioms in generic libraries, we considered the elements of their composition and were able to reduce many program elements to micropatterns [10]. A micropattern is a predicate on a class that can be expressed in terms of its attributes, types, name, or body. We map these idioms onto the micropattern concept for two reasons. First, micropatterns provide a method of encapsulating observations of classes that does not depend on the evaluation of relationships between them. Second, most of idioms we have described can be expressed as predicates on the properties of a class definition. The only exceptions are tag dispatch and constrained function templates, which deserve special attention, and partial template specializations (including `enable_if`), which depend on the analysis of their specialization relationship.

We extended and refined this set of micropatterns, adapting it to C++ and generic programming. The set of new and adapted micropatterns and their descriptions are given in Table 1. We provide a mapping of each micropattern to the idiom that it represents.

We added a new top-level category, *Specialization*, to the micropattern catalog. This category of micropatterns describes properties of class template specializations. The one subgroup we identified, *Degenerate Specializations*, refers to the fact that the specializations are not statically polymorphic with the base template. The names chosen for the micropatterns are intended be representative of the predicate rather than the more established name of the idiom. Specific adaptations and extensions of the previous catalog, and rationale or other notes are now given.

Designator, Taxonomy, and Joiner. We refined these micropatterns, restricting them to non-template classes. Note that under our definition, all taxonomies and joiners are also designators. These micropatterns are tag classes in the generic programming literature.

Traits Class. A traits class is a class template that decouples a generic abstraction from specific implementations. It is comprised entirely of typedefs and occasionally static methods and attributes.

Type, Integral, and Constant Metafunctions. These micropatterns are classes that are used to compute or evaluate properties of types (or integral constants) at compile time. We further note that a class deriving from a metafunction is also a metafunction.

Metafunction Class. A metafunction class is a class (possibly a class template) that contains a nested metafunction. Metafunction classes are frequently used to

delay the instantiation of templates (or partially instantiate them).

Functor. A functor is a class (often a template) that overloads the function call operator in order to interoperate with C++ high-order functions or generic algorithms. Functors are frequently Stateless (i.e., having no side effects) or Environments (i.e., referring to external data sources), depending on their intended purpose.

Function Class. We opted to refer to the "Cobol-Like" design pattern as a "*function class*" since the function is associated with the class rather than its object. This micropattern occurs frequently in generic libraries as a means of deferring the instantiation of a function template, especially in the BGL.

Stateless. A stateless class is one that declares no member variables, but may have static attributes.

Record. A record is a class with no member functions (excluding constructors) and only public non-reference member variables. Records are often used as simple value or POD (Plain old data) types.

Environment. An environment is a class whose attributes are all references (or pointers) to data outside the class itself. The pattern's name is derived from the fact that these classes are frequently used to emulate a function calling environment: a set of arguments and/or accumulated results.

Static Outline. Similar to the concept of a (dynamic) outline, the static outline invokes functionality on its derived classes via static down-casting rather than virtual methods. The name is derived from the fact that the *Outline* micropattern is essentially the abstract framework class in the *Template Method* design pattern [9]. In [8], the Generic Template Method demonstrates how CRTP is used to statically delegate to the derived class. For this reason, we refer to a class know to use CRTP as a *Static Outline*.

Mixin. A mixin is a class that derives from a template parameter. This pattern is used to compose interfaces or data structures.

Enabler. An enabler (or disabler) is a Metafunction with a specialization that is not a metafunction (i.e., it does not define a nested type member). This micropattern is typified by Boost's `enable_if` and `disable_if` classes and is more fully defined in E. A class deriving from an enabler is also an enabler.

Partial Template. A partial template is an empty or undefined class template with one or more non-empty class template specializations. These are sometimes used to restrict the set of types over which a template can be instantiated, especially when that set of types is small.

Identifying instances of tag dispatch and constrained function templates can be done by checking the parameter and return types of functions. Nominally, we could express these as nanopatterns, but since we only identify two such conditions, we have opted to bypass their formalization.

If a function takes a tag class as the type of a function parameter, then it participates in tag dispatch. Likewise if the template name of an enabler is found as a function parameter type or the return type of a function, then it is being actively constrained. Instances where SFINAE is used casually (i.e., not via `enable_if` or `disable_if`) to constrain templates can be very difficult to detect specifically (with a low false positive rate).

V. Implementation and Validation

In order to study the idioms in generic libraries, we extended our srcTools framework [23] to identify the micropatterns described in the previous section. srcTools is a srcML-based [4] fact extraction and source code analysis framework for C++.

srcML is a lightweight, lexical markup for C++ that embeds structural information about the contents of a source file in the output XML file. The approach espoused by srcML is intended to support the rapid construction of source code analysis tools. Because the srcML translator is intended is a reverse engineering parser that aims to support lightweight tools and rapid application development, it forgoes many of the responsibilities traditionally associated with compiler parsers. The srcML translator does not preprocess source code, nor does it perform any semantic analysis. By forgoing these aspects of compilation, the srcML translator is substantially faster and more robust (able to handle a broader set of dialects) than most traditional compilers. Efficiency and usability come at the cost of accuracy. The srcML translator is not able to disambiguate some aspects of the language and can generate inaccurate srcML output.

srcTools is a Python-based framework that supplements the lightweight approach of srcML by implementing more traditional parsing capabilities on top of the srcML format. The srcTools parser component is capable of reconstructing an AST from the XML input generated by the srcML translator. However, srcTools also accepts that source code at "face value", which is to say that it does not preprocess the source code and does not attempt to instantiate templates. Although maintenance of symbol tables and an AST-like model greatly improves the accuracy and efficacy of the parsing and modeling framework, srcML markup errors are still propagated into the output.

In order to compensate for possible markup errors and the lack of preprocessing, the srcTools AST allows for inaccurate type references in its program model. This feature allows us to use srcML+srcTools to reverse engineer any C++ programs without worrying about their external dependencies or even system header files.

The srcTools framework supports application development and extension as a variation of the Observer pattern. This is to say that Python modules interact with the parser framework by registering handlers for specific parsing or AST-construction events. The internal construction of the srcTools AST is actually built using this same mechanism.

A fact extraction front-end to srcTools, the srcfacts program, is used to construct and populate a relational database (SQLite) containing declarations parsed from source code. This database stores each kind of AST element (class, function, method, constructor, etc) in a separate table, indexed by the globally unique name of each element. For example the standard `vector` class template would found in the `src_class` table with the identifier `vector<$T1-1,`

TABLE 3. COUNTS OF MICROPATTERN INSTANCES FROM THE BOOST GRAPH LIBRARY.

Micropattern	Manual	Auto
Designator	112	105
Taxonomy	7	9
Joiner	2	2
Traits Class	58	79
Type Metafunction	159	111
Integral Metafunction	73	21
Constant Metafunction	57	68
Metafunction Class	39	50
Functor	176	171
Function Class	31	32
Stateless	522	596
Record	34	28
Environment	48	51
Static Outline	4	2
Mixin	11	11
Enabler	0	6
Partial Template	17	14

TABLE 2. ACCURACY, PRECISION, AND RECALL FOR EACH OF THE MICROPATTERN.

Micropattern	Acc.	Prec.	Rec.
Designator	.98	.96	.90
Taxonomy	1.0	.78	1.0
Joiner	1.0	1.0	1.0
Traits Class	.97	.67	.91
Type Metafunction	.94	.95	.67
Integral Metafunction	.94	1.0	.29
Constant Metafunction	.99	.78	1.0
Metafunction Class	.99	.84	1.0
Functor	.99	.99	.97
Function Class	1.0	.97	1.0
Stateless	.91	.87	.99
Record	.98	.86	.71
Environment	.99	.92	.98
Static Outline	1.0	1.0	.50
Mixin	1.0	1.0	1.0
Enabler	.99	0.0	NaN
Partial Template	.99	.93	.76

`$T1-2>`, with `$T` indicating a type parameter, and the *m-n* notation describing its canonical position rather than name. The second type parameter is an allocator. This particular naming of class and function templates enables srcTools to effectively differentiate templates and their specializations. For example, the `vector<bool>` specialization is encoded as `std::vector<bool, $T1-1>`. This technique was pioneered in our previous work [23].

In order to support his work, we developed a new idiom-identification module for srcTools and integrated it into the srcfacts fact extractor. This module augments the existing database with a table containing instances of identified micropatterns. Each row in this table contains the unique identifier of a class or class template and a sequence of Boolean values indicating whether or not the class was identified as any of the 17 micropatterns given in Table 1.

To help demonstrate the viability of this software to conduct large-scale empirical studies, we conducted a controlled experiment to determine the accuracy, precision, and recall of the srcfacts tool and the micropattern identification module. We evaluated the tool against the Boost Graph Library (BGL) v1.41.0 [19], which is part of the Boost C++ Libraries[1]. The BGL is 56 KSLOC[2], and has approximately 990 classes (not counting class template specializations). More importantly, the BGL is known to be "heavily generic" and includes generic data structures, algorithms, and a substantial amount of template metaprogramming. We manually examined each of the classes and classified them according to the micropatterns described in Section IV. Counts of these observations are given in Table 4. The srcfacts tool identified 922 of the classes (about 93%).

Of these 922 classes, we observed the micropattern counts in 85% of them. We can interpret this as a measure of the degree of "idiomization" within these libraries. The remaining 15% are comprised of more traditional (often generic) data structures, describing objects exhibiting both state and behavior.

There are two interesting observations to make regarding these counts. First, the number of stateless classes is quite high, but this largely reflects the set of classes that are designators, taxonomies, joiners, metafunctions, a large number of functors, and partial templates. Second, we manually identified no enablers in the BGL. This is because the BGL relies on the Boost's `enable_if` template, which is defined outside this body of source code.

The precision, recall and accuracy of these counts are shown in Table 2. Accuracy, precision, and recall are defined as ratios of true and false positives or negatives.

$$accuracy = \frac{tp + tn}{tp + tn + fp + fn}$$

$$precision = \frac{tp}{tp + fp} \qquad recall = \frac{tp}{tp + fn}$$

We note that the recall for identifying Enablers is undefined. Since we manually identified no instances of this micropattern, there can be no false negatives. On average srcfacts identifies instances with 98% accuracy, 85% precision, and 85% recall (excluding the count for Enablers). As a result, we feel fairly confident in the ability of this tool to identify idiom instances in generic libraries.

There are naturally threats to the validity of this study. First, there may be errors in the manual classification of templates in the BGL. To help reduce the error rate, we cross-checked the manual results with the automated results and investigated every disagreement. Second, the srcTools framework is designed for imprecise parsing to accommodate partial, incomplete, and un-preprocessed source code. As such, the parser will often fail to construct accurate models of the source code. Also, these measurements are made with respect to classes identified both manually and automatically; we do not include the 70 or so classes not identified by srcfacts.

[1] See http://www.boost.org/ for details.

[2] All SLOC counts generated using David A. Wheeler's SLOCCount. See http://www.dwheeler.com/sloccount/ for details.

VI. An Empirical Study of Generic Libraries

As part of this work, we have used srcfacts to study the occurrence of these idioms within a number of well-know generic libraries, specifically the Standard C++ Library (GCC-4.4.3 [3]), the Computational Geometry Algorithms Library (CGAL-3.5[4]), and the Boost C++ Libraries (1.41.0). Note that the Boost C++ libraries are actually a collection of (sometimes largely) independent generic and systems libraries. In total, we surveyed approximately 1.1 MSLOCs and just fewer than 26,000 classes. Library percentages of micropattern identified by srcfacts are shown in Table 4.

TABLE 4. SLOCs, number of classes, and micropattern count percentages from GCC's Standard C++ Library, CGAL, and the Boost C++ Libraries.

Micropattern	GCC Std	CGAL	Boost
SLOC	70,200	436,288	781,979
Classes	149	4,572	21,237
Designator	4.7%	2.5%	3.2%
Taxonomy	2.0%	0.4%	0.5%
Joiner	0.0%	0.1%	0.4%
Traits Class	14.1%	8.7%	7.1%
Type Metafunction	0.0%	1.6%	15.6%
Integral Metafunction	0.0%	0.1%	0.7%
Constant Metafunction	0.0%	0.0%	0.5%
Metafunction Class	0.0%	0.0%	1.3%
Functor	16.8%	32.5%	4.5%
Function Class	7.4%	0.3%	3.0%
Stateless	64.4%	54.0%	52.3%
Record	7.4%	1.6%	2.5%
Environment	14.8%	4.3%	2.5%
Static Outline	0.0%	0.0%	0.1%
Mixin	0.0%	2.3%	0.8%
Enabler	0.0%	0.0%	0.1%
Partial Template	0.0%	1.5%	1.6%

The most outstanding result from this study is the large percentage of stateless classes found in C++ generic libraries. Here, the percentage of classes is 64%, 54%, and 52% for the Standard C++ Library, CGAL, and Boost, respectively. In contrast Java (object-oriented) products contain 6-15% stateless classes [10]. This is easily attributed to the large number of metafunctions, traits classes, and partial templates found in these libraries. Only the Boost C++ Libraries seem to contain significant numbers of metafunctions.

Functors also contribute to the high stateless counts. We find that 80%, 85%, and 71% of all functors are stateless between the Standard Library, CGAL and Boost, respectively. Interestingly, the remaining 15-30% are stateful, with 6 and 10% of functors being identified as environments. This indicates that parameterization over stateful or referential functors is a common practice.

We note that the number of enabler instances is vanishingly small (0 in the Standard Library, 1 in CGAL and 25 in Boost). This is generally attributable to the fact that a single enabler/disabler can simply be reused throughout the library. Because Boost libraries are relatively independent of each other, it is not uncommon for idioms to be duplicated.

We also searched the acquired data for uses of the `enable_if` and `disable_if` to constrain function templates and found that their use is practically non-existent. In Boost, only 182 of 34,684 functions, methods, or constructors referenced those data types. We can attribute this lack of explicit constraints to a) the detriment to readability caused by the use of enablers on functions templates, b) the use of concept checking libraries within template definitions [20, 27]. In fact, feature only seems to be used when instantiation with the wrong type can lead to subtle type-based runtime errors.

Additionally, we conducted an informal survey of tag classes and their usage. In this investigation, we discovered that designators are frequently used to represent disjoint or boolean properties of types, taxonomies are used to represent categorized properties, and joiners are used to merge multiple, orthogonal categories (taxonomies) into a single property. There are few categories represented in these libraries. We suspect that the number of joiners is actually much higher than reported due to incomplete information during parsing. We further observed that only 11% of tag classes are used as function parameters. From this, we infer that tag dispatch is not a commonly used technique.

VII. Discussion

We begin the discussion by observing that new tools and techniques are needed to support the comprehension, construction, and maintenance of C++ generic libraries. The benefit of research and development in the areas of reverse engineering, program comprehension, and software maintenance have been of great benefit to practitioners in the areas of other software development paradigm, especially object-oriented programming.

The object-oriented paradigm is of particular interest because it provides the "language" in which the surveyed idioms are mostly written. By this we mean that most of the programming idioms surveyed and identified are rooted in the language of object-oriented programming: classes. However, our survey indicates that the construction techniques for generic libraries are definitely not object-oriented in nature. Most of the classes used to support the generic paradigm do not describe objects. Our study indicates that the non-object-oriented components of the libraries make up a vast majority of the libraries' composition. A direct result of this "couching of one language within another" is the misleading results that can be produced by existing reverse engineering tools.

In this paper, we present a tool that is capable of automatically identifying, with some margin of error, the kinds of idioms being used in the construction of generic libraries. The abstract labeling or stereotyping of elements is often used to improve the ways in which developers interact with source code [7]. While this technique might be used to inform a developer of what a class might actually represent, the tool cannot help determine the role of the idiom in the design of the library. Clearly, more work is needed in the area of design recovery for generic libraries. This is not possible without first understanding how these programming idioms relate to the design of such libraries. Many of these

[3] See http://gcc.gnu.org/ for details.
[4] See http://www.cgal.org/ for details.

idioms surveyed in this paper are closely associated with the evolution of the C++0x programming language. One well-developed and obvious example of this association is that of functors to C++0x lambda functions.

Traits classes, metafunctions, tag dispatch, and template constraints are closely related to *concepts*, an extensive set of language features proposed C++0x [6, 11, 21]. Traits classes and metafunctions (type traits) are very similar to the abstractions that can be described by concept definitions, and the tag class hierarchies (taxonomies and joiners) are used to define properties and categories of types. The use of `enable_if` can be deprecated through features for explicitly constraining templates. However, despite the potential for improvement in the readability and writeability of generic libraries, concepts were removed from the C++0x proposal. We can easily imagine, however, that an effective definition of concepts for C++ will ameliorate many of the complexities leaked through generic libraries, while creating a number of interesting opportunities for tool builders.

VIII. CONCLUSIONS

We have presented a survey of common programming idioms used in the construction of C++ generic libraries, and implemented a tool that is capable of identifying instances of these idioms in terms of micropattern instances. The techniques and tools discussed in this paper are intended to support a programmer's comprehension of C++ generic libraries. Based on evaluation, the approach can correctly identify the elements of design and implementation used in the construction of these libraries—elements that, despite being rooted in the syntax of an object-oriented construct (i.e., classes), are decidedly atypical of object-oriented construction. We feel that this is a good first step toward addressing comprehension problems in this domain.

REFERENCES

[1] D. Abrahams and A. Gurtovoy, C++ template metaprogramming: Concepts, tools, and techniques from boost and beyond: Addison Wesley, 2005.

[2] A. Alexandrescu, Modern c++ design: Generic programming and design patterns applied: Addison Wesley, 2001.

[3] M. Austern, Generic programming and the stl: Using and extending the c++ standard template library, 7th ed. Boston, Massachusetts: Addison-Welsey Longman, 1998.

[4] M. L. Collard, H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, Oregon, May 10-11, 2003, pp. 134-143.

[5] J. Coplien, "Curiously recurring template patterns," C++ Report, vol. 7, pp. 24-27, Feb, 1995.

[6] G. Dos Reis and B. Stroustrup, "Specifying c++ concepts," in 33rd Symposium on Principles of Programming Languages (POPL'06), Charleston, SC, Jan 11-13, 2006, pp. 295-308.

[7] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse engineering method stereotypes," in 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, Pennsylvania, Sep 25-27, 2006, pp. 24-34.

[8] A. Duret-Lutz, T. Géraud, and A. Demaille, "Design patterns for generic programming in c++," in 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01) San Antonio, Texas, 2001, p. 14.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns elements of reusable object-oriented software: Addison Wesley, 1995.

[10] J. Y. Gil and I. Maman, "Micro patterns in java code," in 20th Conference on Object-Oriented Programming Systems, Langauges, and Applications (OOPSLA'05), San Diego, California, Oct 16-20, 2005, pp. 97-116.

[11] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, "Concepts: Linguistic support for generic programming in c++," in ACM SIGPLAN Conference on Object-Oriented Programming, Systemls, Languages, and Applications (OOPSLA'06), Portland, Oregon, Oct 22-26, 2006, pp. 291-310.

[12] D. Gregor and S. Schupp, "Stllint: Lifting static checking from languages to libraries," Software: Practice and Experience, vol. 36, pp. 225-254, Mar, 2005.

[13] R. Holeman, "Identifying programming idioms in c++ generic libraries," in Department of Computer Science. vol. MS: Kent State University, 2009, p. 72.

[14] J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock, "An analysis of constrained polymorphism for generic programming," in Workshop on Multiparadigm Programming with Object-Oriented Languages, Anaheim, California, Oct 26, 2003, pp. 87-107.

[15] J. Järvi, J. Willcock, and A. Lumsdaine, "Concept-controlled polymorphism," in 2nd International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany, Sep 22-25, 2003, pp. 228-244.

[16] D. Musser and A. Stepanov, "A library of generic algorithms in ada," in SIGAda International Conference on Ada, Boston, Massachusetts, Dec, 1987, pp. 216-225.

[17] D. Musser and A. Stepanov, "Algorithm-oriented generic libraries," Software: Practice and Experience, vol. 24, pp. 623-642, Jul, 1994.

[18] Z. Porkoláb, J. Mihalicza, and Á. Sipos, "Debugging c++ template metaprograms," in 5th International Conference on Generative Programming and Component Engineering (GPCE'06), Portland, Oregon, Oct 22-26, 2006, pp. 255-264.

[19] J. Siek, L.-Q. Lee, and A. Lumsdaine, The boost graph library: User guide and reference manual: Addison-Wesley, 2001.

[20] J. Siek and A. Lumsdaine, "Concept checking: Binding parametric polymorphism in c++," in 1st Workshop on C++ Template Programming, Erfurt, Germany, Oct 10, 2000.

[21] J. Siek and A. Lumsdaine, "Essential language support for generic programming," in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05), Chicago, Illinois, Jun 11-15, 2005, pp. 73-84.

[22] Y. Smaragdakis and D. Batory, "Mixin-based programming in c++," in Generative and component-based software engineering. vol. 2177: Springer Berlin/Heidelberg, 2001, pp. 164-178.

[23] A. Sutton, R. Holeman, and J. I. Maletic, "Abstracting the template instantiation relation in c++," in 25th International Conference on Software Maintenance, Edmonton, Canada, Sep 20-26, 2009, pp. 559-562.

[24] A. Sutton and J. I. Maletic, "Automatically identifying c++0x concepts in function templates," in 24th International Conference on Software Maintenance (ICSM'04), Beijing, China, Sep 28-Oct 4, 2008, pp. 57-66.

[25] M. VanHilst and D. Notkin, "Using c++ templates to implemnt role-based designs," in Object technologies for advanced software. vol. 1049: Springer Berlin/Heidelberg, 1996, pp. 22-37.

[26] M. Zalewski and S. Schupp, "Change impact analysis for generic libraries," in 22nd Conference on Software Maintenance (ICSM'06), Philadelphia, PA, Sep 24-27, 2006, pp. 35-44.

[27] I. Zólyomi and Z. Porkoláb, "Towards a general template introspection library," in 3rd International Conference on Generative Programming and Component Engineering (GPCE'04), Vancouver, Canada, Oct 24-28, 2004, pp. 266-282.