UNDERSTANDING AND MAINTAINING C++ GENERIC LIBRARIES

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by
Andrew Sutton
August, 2010
Dissertation written by

Andrew Sutton

B.S., Ohio University, USA, 1999

M.S., Kent State University, USA, 2005

Ph.D., Kent State University, USA, 2010

Approved by

Dr. Jonathan Maletic
Chair, Doctoral Dissertation Committee

Dr. Gwenn Volkert
Members, Doctoral Dissertation Committee

Dr. Mikhail Nesterenko

Dr. Michael Collard

Dr. Donna Witter

Accepted by

Dr. Robert A. Walker
Chair, Department of Computer Science

Dr. Timothy Moerland
Dean, College of Arts and Sciences
# TABLE OF CONTENTS

TABLE OF CONTENTS ........................................................................................................... III

LIST OF FIGURES ............................................................................................................... VIII

LIST OF TABLES ................................................................................................................ XVII

ACKNOWLEDGEMENTS ................................................................................................. XIX

CHAPTER 1 INTRODUCTION ............................................................................................ 21

1.1 Research Overview ................................................................................................. 23

1.2 Contributions .......................................................................................................... 23

1.3 Broader Impacts ...................................................................................................... 25

1.4 Organization ............................................................................................................ 26

CHAPTER 2 TEMPLATES + CONCEPTS = GENERIC LIBRARIES ............................. 28

2.1 Templates ................................................................................................................ 28

2.1.1 Function Templates ............................................................................................ 29

2.1.2 Class Templates ................................................................................................. 33

2.1.3 Non-Type and Variadic Template Parameters .................................................. 39

2.2 Concepts ................................................................................................................ 47

2.2.1 Concept Definitions .......................................................................................... 48

2.2.2 Constrained Templates .................................................................................... 53

2.2.3 Concept Checking ............................................................................................. 56

2.2.4 An Alternative Syntax ...................................................................................... 59

2.3 Generic Libraries ..................................................................................................... 60
4.3 Discussion ........................................................................................................... 98
  4.3.1 The Nature of Concepts ................................................................................. 98
  4.3.2 Casual Modeling ............................................................................................ 99
4.4 Conclusions ........................................................................................................ 101

CHAPTER 5 REVERSE ENGINEERING GENERIC SOURCE CODE ............ 103
  5.1 Related Work ................................................................................................... 103
  5.2 Reverse Engineering Templates and Concepts ................................................ 105
    5.2.1 An Program Model for Templates and Concepts ...................................... 105
    5.2.2 Implementation ......................................................................................... 110
  5.3 Source Code Models for Generic Libraries ...................................................... 112
    5.3.1 Template Instantiation Graph ..................................................................... 113
    5.3.2 Constraint Propagation Graph ..................................................................... 117
    5.3.3 Concept Lattice .......................................................................................... 118
  5.4 Conclusions ....................................................................................................... 120

CHAPTER 6 IDIOM USAGE IN GENERIC LIBRARIES ............................ 121
  6.1 Micropatterns for Generic Libraries ................................................................. 121
  6.2 Validation ......................................................................................................... 126
  6.3 Empirical Study ............................................................................................... 130
  6.4 Discussion and Conclusions ............................................................................. 132

CHAPTER 7 A REFERENCE ARCHITECTURE FOR GENERIC LIBRARIES135
  7.1 Architectural Description ................................................................................. 135
    7.1.1 Conceptual Elements ................................................................................. 138
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.2</td>
<td>Behavioral Elements</td>
<td>138</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Structural Elements</td>
<td>141</td>
</tr>
<tr>
<td>7.2</td>
<td>Validation</td>
<td>144</td>
</tr>
<tr>
<td>7.3</td>
<td>Conclusions</td>
<td>149</td>
</tr>
<tr>
<td><strong>CHAPTER 8</strong> MIGRATING TEMPLATES TO CONCEPTS</td>
<td></td>
<td>151</td>
</tr>
<tr>
<td>8.1</td>
<td>Overview</td>
<td>152</td>
</tr>
<tr>
<td>8.2</td>
<td>Approach</td>
<td>154</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Requirement Extraction</td>
<td>155</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Concept Matching</td>
<td>156</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Concept Instantiation</td>
<td>159</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Refinement Search</td>
<td>163</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Concept Analysis</td>
<td>165</td>
</tr>
<tr>
<td>8.3</td>
<td>Evaluation</td>
<td>170</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Applicability</td>
<td>171</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Stability</td>
<td>173</td>
</tr>
<tr>
<td>8.4</td>
<td>Discussion</td>
<td>175</td>
</tr>
<tr>
<td><strong>CHAPTER 9</strong> DIAGNOSING CONCEPTUAL DESIGN FLAWS</td>
<td></td>
<td>177</td>
</tr>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>177</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Frequently Co-Occurring Constraints</td>
<td>178</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Ambiguous Constraints</td>
<td>181</td>
</tr>
<tr>
<td><strong>CHAPTER 10</strong> CONCLUSIONS</td>
<td></td>
<td>185</td>
</tr>
<tr>
<td><strong>APPENDIX A</strong> CONCEPTS FOR C++0X</td>
<td></td>
<td>189</td>
</tr>
</tbody>
</table>
A.1 Support Concepts ................................................................. 189
A.2 Standard Concepts ............................................................ 191
A.3 Iterator Concepts .............................................................. 202

REFERENCES ........................................................................... 205
LIST OF FIGURES

Figure 1. The swap function template exchanges the values of two objects............. 29

Figure 2. The instantiated swap function with substitution for the type int. The
    template keyword and swap<int> identifiers indicate a template specialization.

Figure 3. A specific version of File swap exchanges underlying file handles without
    relying on copy or assignment semantics.............................................. 31

Figure 4. A specialized List swap exchanges only the head and tail pointers of lists
    rather than copying all of the elements in both lists.................................... 31

Figure 5. The member swap specialization relies on SFINAE to disable the overload for
    types that do not have a member swap...................................................... 32

Figure 6. An outline of a generic Vector data structure is parameterized over its
    contained object type T and provides a number of member functions and associated
    types........................................................................................................... 34

Figure 7. Declaring and using the Vector causes the compiler to instantiate the
    template........................................................................................................ 35

Figure 8. The Vector equality comparison operator performs a pairwise equality
    comparison of equality the elements of this and another Vector.................. 36

Figure 9. An outline of a vector specialization for boolean values encodes each value
    as a bit rather than an integral representation........................................... 37
Figure 10. The Compact_Value class encodes information about a type T and its minimum bitwise encoding width N.

Figure 11. A partial template specialization of the Vector over the Compact_Vector class template can be used to implement space-optimal vectors for integral types T with minimum bit encodings of width N.

Figure 12. The Array wrapper provides uniform C++ semantics (e.g., copyability) for C arrays. It is parameterized over a type and size.

Figure 13. The Transform class template defines a wrapper functor for all unary functions on a type T.

Figure 14. The Stack adaptor is parameterized over a type T and the name of a container data structure, which defaults to the List template.

Figure 15. The List_Node class uses forwarding constructor to construct its link structure and underlying data object.

Figure 16. The construction of a new List_Node forwards a copy of the given object to the node’s constructor, which is in turn forwarded to T’s copy constructor.

Figure 17. The Tuple data type is a recursively defined variadic template with two primary specializations. The specific implementation is omitted.

Figure 18. A partial template specialization of Tuple that causes the data structure to be self-flattening.

Figure 19. Declarations of variadic non-type and template template parameters.

Figure 20. Concept definitions for Iterator and Input_Iterator types express required interfaces and behaviors.
Figure 21. The concept hierarchy defines a taxonomy of iterators based on their supported operations and semantics. ................................................................. 51

Figure 22. A concept map adapts pointer types to the Random_Access_Iterator concept................................................................. 52

Figure 23. The find algorithm requires that the Iter template parameter satisfy the requirements of Input_Iterator and that an operator== overload exists for the iterator’s value_type and T................................................................. 53

Figure 24. The distance operation is overloaded on the Iterator and Random_Access_Iterator concepts, allowing the compiler to differentiate constant and linear time implementations based on the kind of iterators.............. 54

Figure 25. A partial listing of the Pair class template and its constraints. Template parameters T and U are required to model the Variable_Type concept .......... 55

Figure 26. Two range constructors for the Vector data type are differentiated by the kinds of Iterators. ................................................................. 57

Figure 27. A classification of types different basic Object_TYPES from Classes and Unions................................................................. 58

Figure 28. The Iterator concept written using valid expressions............... 59

Figure 29. The concept assertion is roughly analogous to a concept map, and explicitly adapts a type to a concept................................................................. 60

Figure 30. The greater functor encapsulates the comparison, via operator>, for a type T. ................................................................................. 64
Figure 31. The is_sorted function is a high-order function parameterized over a binary relation, Comp.

Figure 32. The is_sorted algorithm is called using a lambda expression to define and instantiate an anonymous functor.

Figure 33. The is_same type trait returns true if the two types are the same and false otherwise.

Figure 34. The result_of type trait allows programmers to access the result a function type or functor.

Figure 35. The iterator_traits class decouples the types associated with iterators from their implementation. The partial specialization supplies the requisite type names for pointer types.

Figure 36. A tag class hierarchy defines a taxonomy of iterator kinds, and a type accessor provides a method of determining an iterator’s kind.

Figure 37. The distance operation uses tag dispatch to select an optimal implementation based on the kind of iterator.

Figure 38. The member_swap traits determine if a type T defines a member variable named swap.

Figure 39. The partial template container_gen must be parameterized over a valid container selector (Tag) class and value type.

Figure 40. The enable_if template defines a result type if the boolean predicate is true. No nested type is defined otherwise.
Figure 41. The sort algorithm uses concept-controlled polymorphism to select either the canonical swap algorithm or a member-specialized variant. ........................................ 76

Figure 42. The bool_ class template is used to represent compile-time boolean values as constant metafunctions. ........................................................................................................ 77

Figure 43. The not_ metafunction defines a method of computing the logical not of boolean metafunction constants true_ and false_. ......................................................... 78

Figure 44. The variadic and_ metafunction computes the logical conjunction of its arguments. Each argument is required to be a boolean metafunction. ......................... 79

Figure 45. The default constructible archetype uses the mixins pattern to support interface aggregation. ........................................................................................................ 80

Figure 46. 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... 82

Figure 47. The Same_Type concept is implemented as a class template with a nested check metafunction and assertion class. ................................................................. 85

Figure 48. A definition of the explicit concept Forward_Iterator is always invalid, unless explicitly declared to be otherwise. ................................................................. 86

Figure 49. The Forward_Iterator<T*> specialization denotes the satisfaction of the Forward_Iterator concept by all pointer types......................................................... 87

Figure 50. The Equality_Comparable concept requires T to support comparisons via operator== and operator!=. ................................................................. 89
Figure 51. The Has_Plus concept contains a deduced type result for the addition of types $T$ and $U$. 

Figure 52. The Has_Virtual_Destructor concept class refines HasDestructor and Polymorphic_Class and provides specialized concept-checking logic in its check and assertion components.

Figure 53. The requirements for the Semiregular concept are implemented as a nested, named requirement that is checked similarly to its refinements.

Figure 54. A constrained implementation of the find algorithm implements its constraints as a default template parameter.

Figure 55. The three overloads of the iterator advance function. Overload differentiation is done using filtering expressions on the iterator hierarchy as opposed to tag dispatch.

Figure 56. The Origin C++0x Libraries are modeled on the C++ Standard Library and includes components for template metaprogramming, type traits, and concepts at its core.

Figure 57. The Forward_Iterator concept is decomposed into two concepts: Multipass_Iterator and Forward_Iterator.

Figure 58. The C++0x program model that supports template elements and template specialization.

Figure 59. The subset of the C++0x program model that supports concepts, concept maps, and refinement.
Figure 60. A subset of the C++0x program model that describes generic program elements. ................................................................. 108

Figure 61. The names of template parameters are allowed to vary between declarations and definitions or omit them entirely. ................................................................. 108

Figure 62. The extract, wrap, and normalize metafunctions are used to determine the instantiation of vector_base for the given vector specialization. .......... 114

Figure 63. The template instantiation graph shows the dependencies between classes on templates. ................................................................................................. 115

Figure 64. The rebind_allocator metafunction uses the extract metafunction to retrieve an allocator type, and then rebinds it to the type T. ......................... 115

Figure 65. The instantiation graph of the rebind_allocator metafunction instantiates both extract_allocator and the rebind template of the type variable A. ........................................................................................................ 116

Figure 66. The constraint propagation graph for the Input_Iterator concept shows the propagation of type requirements across concept constraints. ..................... 118

Figure 67. The concept lattice for the C++ type system defines the basic kinds of types and type constructors available in the C++0x programming language. .......... 119

Figure 68. Client applications instantiate and use generic algorithms, data structures, and adaptors. Concepts describe and constrain the interoperability of algorithms and data types................................................................. 136
Figure 69. The adjacent_find algorithm returns requires the type Iter to be a Forward_Iterator and that its associated value_type be Equality_Comparable.

Figure 70. The generic swap algorithm exchanges the value of two objects. A specialization for list<T> optimizes the general case.

Figure 71. The vector range constructors are restricted by the kinds of iterators over which they are instantiated.

Figure 72. A partial listing of the vector class template provides access to a range of iterators (pointers).

Figure 73. The reverse_iterator adaptor reverses the traversal direction of any iterator that models the Bidirectional_Iterator concept.

Figure 74. A clustering of Boost C++ Libraries by (cosine) similarities, using flexible linking, and based on their composition of architectural elements. The threshold marker (0.32) denotes top-level clusters of in terms of architectural composition. Clusters are labeled for convenience.

Figure 75. A simple implementation of the generic std::fill algorithm.

Figure 76. A new definition for the fill algorithm that lists requires the satisfaction of concept instances in order to be instantiated.

Figure 77. A partial listing of the preprocessed OutIter concept definition.

Figure 78. The Has_Equal concept takes two parameters, but the default value of the second is given as the first.
Figure 79. The Has_Constructor concept requires a type $T$ to be constructible over a sequence of argument types. ................................................................. 162

Figure 80. The refinement lattice for the Iter-based concept instances. Concept names are abbreviated, and initially matched concepts are shown in gray. ......................... 164

Figure 81. The coverage lattice generated for the concept instances and type requirements in the fill algorithm. In this representation, concept names are abbreviated, function names represented symbolically, and same-type constraints are hidden. ................. 168

Figure 82. Two minor variations in the implementation of the fill algorithm. The first compacts a number of expressions in the original into a single statement, and the second expands them into multiple statements. ...................................................... 174

Figure 83. A simple implementation of the fill algorithm. ................................. 180

Figure 84. An OutputIterator concept definition that expresses a semantic requirement for its reference type to be assignable to its value type. ......................... 180

Figure 85. The previous iterator concept hierarchy defined by the C++ Standard Library. Arrows denote the direction of refinement. ............................... 182

Figure 86. The new OutputIterator expresses an assignment requirement between an Iterator’s reference type and a value type. ............................... 183
LIST OF TABLES

Table 1. A mapping of C++0x concept features to programming idioms used in modern C++ generic libraries ................................................................. 62

Table 2. 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 ........................................................................ 122

Table 3. Counts of micropattern instances from the Boost Graph Library ................ 127

Table 4. Definitions of accuracy, precision, and recall are computed in terms of true/false positives and negatives ............................................................. 128

Table 5. Accuracy, precision, and recall for each of the micropatterns observed in the Boost Graph Library ......................................................................... 129

Table 6. SLOCs, number of classes, and micropattern count percentages from GCC’s Standard C++ Library, CGAL, and the Boost C++ Libraries ..................... 130

Table 7. Kinds of generic algorithms differentiated by their use of overloading .......... 139

Table 8. The conceptual, structural, and behavioral composition of 66 Boost C++ Libraries by percentage ................................................................................. 145

Table 9. Expressions and their associated requirements extracted from the fill algorithm ........................................................................................................ 156

Table 10. The initial set of concepts that match the type requirements extracted from the fill algorithm .................................................................................... 157
Table 11. An inversion of the mapping in Table 10, concepts are mapped to the set of requirements that they may express................................................................. 158

Table 12. The initial set of requirements extracted for the fill algorithm............. 162

Table 13. The initial set requirements generated from the refinement lattice search.... 165

Table 14. The relation mapping concept instances to requirements of the fill algorithm. Concept arguments and types are omitted for brevity. ......................... 167

Table 15. A selected listing of STL algorithms, their computed concept requirements (with abbreviated names). The requirements of each algorithm is described by a conjunction of concept instances with ambiguities written as disjunctions. .......... 171

Table 16. Concept requirements for minor variations on the same algorithm......... 174
ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Jonathan Maletic, for allowing me the time and freedom to pursue my own interests and research directions. His accommodation of my interests in computer programming allowed me to develop the work that has become this dissertation. Of equal importance was his advice and guidance on the development and presentation of research, the results of which are also evident in this work.

This work would also not have been possible without the support of Dr. Michael Collard and his work on srcML. It was the basis for all of the technical contributions in this dissertation and also my Master’s Thesis. I would also like to thank the rest of my dissertation committee for the feedback received on the presentation of this dissertation.

During this work, I was supported by Google through their Summer of Code program in 2007 and 2008. I owe a special ‘thank you’ to Dr. Jeremy Siek for introducing me to the style of generic programming that ultimately resulted in this work.

A number of students at Kent State also deserve thanks for their work on several of my side projects involving the Origin C++0x Libraries and acting as “sounding boards” for ideas on generic library design and language features for concepts. In no particular order these are: Michael Lopez, Thomas Mullaly, Brian Bartman, and Chris Wagner. I wish them luck in their careers.
Most importantly, I would like to thank my wife, Michelle for her unending patience with my graduate tenure. Sorry it took so long. The rest of my family—Mom, Dad, Tyler, Gail, Mike, and Meryl—deserve thanks for their support and advice. I would also like to thank the menagerie: Beaker, who seems to be bipolar, Gato, who is fat, and Roxanne, who couldn’t be here, but loved her Grinch.

Finally, and capriciously, I thank Northeast Ohio for being an often gloomy but otherwise not half bad place to live.

Andrew Sutton

May 2010, Kent, Ohio
CHAPTER 1

INTRODUCTION

Generic programming is a discipline that equips developers with a mechanism for designing adaptable and efficient generic algorithms and data structures. The goal of this discipline is to “express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction” [Jazayeri, Loos, Musser 1998]. The adaptability and interoperability of generic data structures and algorithms is most frequently accomplished through parameterization over different types or functions, thus making the generic components more general, reusable, and broadly applicable [Denhert, Stepanov 1998]. This being the case, generic programming can be described as the process of writing software that operates on generic parameters. Broadly stated, the principles of generic programming include the separation of concerns, specification of abstraction, and the retention of efficiency. Generic programming is not, per se, independent of other approaches such as functional or object-oriented programming, but is a supplemental methodology for building generic algorithms and data structures within those paradigms.

In C++, generic programming is rooted in the parameterization of algorithms and data structures using templates and the provision of user-specified types at compile time in order to compose more complex software abstractions. Because C++ programs are “close to the metal”, which is to say that they have direct access to memory and processors, they offer unsurpassed runtime efficiency. The templated components of
such these libraries provide convenient abstractions, but generally do not incur the overhead of languages running in interpreters (e.g., JavaScript, Python, Perl, or Ruby) or virtual machines (e.g., Java and C#) because of the compositional nature of templates and derived compiler techniques such as inlining and copy elision. While extensive use of the compiler to generate code can yield substantial benefits in terms runtime performance and elegant software design, the resulting software can be 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 discipline, 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 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 catastrophic cases, runtime errors. This is to say that the potential for error in the compile-time composition of programs is compounded as the composed programs grow in size or depth of composition. Because of the potentially complex and leaky nature of generic libraries, new techniques and tools must be developed to support programmers working in this domain.

1.1 Research Overview

The goals of this research are two-fold: to provide basic mechanisms and models to support the understanding of the C++ generic libraries in the abstract, and to define new tools and techniques to support the software maintenance of C++ generic libraries. The first goal is accomplished by developing a thorough understanding of the language features and programming idioms used that support the programming paradigm and their corresponding responsibilities and abstractions [Sutton, Holeman, Maletic 2010a; Sutton, Maletic 2010]. This knowledge is used to define source code models and analyses to support the reverse engineering, design recovery, and comprehension of C++ generic libraries [Sutton, Holeman, Maletic 2009; 2010b]. The second goal is achieved by developing a novel technique for identifying type constraints in unconstrained function templates [Sutton, Maletic 2008], which directly supports the migration of modern C++ to C++0x using concepts.

1.2 Contributions

The primary contributions presented in this dissertation are, broadly, the study and description of C++ generic libraries and the development of new source code analysis
techniques for evolving constraints and maintain concept hierarchies. Specific contributions to the study and description of generic libraries include:

- the development of a catalog of programming idioms for C++ generic libraries and the definition of a mapping of these programming idioms to proposed C++0x language features [Sutton, Holeman, Maletic 2010b],

- the implementation of a concept emulation library to validate this mapping and provide a framework for experimenting with generic library design [Sutton, Maletic 2010],

- the definition of C++ micropatterns to support the identification of programming idiom instances in C++ generic libraries [Sutton, Holeman, Maletic 2010a],

- the definition and implementation of reverse engineering techniques, tools, and source code models to enable reverse engineering, fact extraction, and design recovery for C++ generic libraries [Sutton 2005; Sutton, Holeman, Maletic 2009; Sutton, Maletic 2007a; b],

- the application of these techniques to conduct empirical studies on generic libraries and a description of their composition [Sutton, Holeman, Maletic 2010a], and

- the derivation of a reference architecture for C++ generic libraries to support program comprehension [Sutton, Holeman, Maletic 2010b].

Specific contributions to the development reverse engineering and source code analysis techniques to support software maintenance include:
• the definition of a new source code analysis technique that directly supports the migration of unconstrained C++ function templates to use C++0x concepts and the maintenance of those constraints [Sutton, Maletic 2008], and
• the use of that technique to experimentally identify and diagnose problems in the specification of constraints and the design of concept hierarchies.

Additionally, the dissertation includes discussion on techniques for the design of concepts and concept hierarchies for generic libraries. However, the efficacy of these techniques cannot be validated since no viable compiler exists to test them.

1.3 Broader Impacts

The work presented in this dissertation will directly support the enhancement and quality of C++ generic libraries through improved program comprehension and software evolution methods and tools. The C++ programming language is deployed in the international networking and data infrastructures, nearly every business, government office, and hospital, our home computing and entertainment systems, our mobile phones and personal computing devices, and in our transportation system and automobiles. The improvements engendered by the research presented in this dissertation will be reflected in the quality and reliability of the many critical and real-time systems that function as the backbone of our daily lives and our information-age society.

The results of the research can be leveraged to help globally institutionalize generic programming and library design in C++ by making these methods and technologies more accessible to programmers. Widespread use and experience with the underlying techniques will support the development of an effective feedback loop (i.e., programmers
to language designers and back again), which will influence the future design of the C++ programming language.

Additionally, aspects of the work presented in this dissertation can be leveraged to develop new pedagogical methods for the teaching of generic programming, and generic library design especially in the areas of algorithms and data structures. The work also supports the extension of software engineering curricula to address issues, techniques, and tools in the software maintenance and evolution of generic libraries.

1.4 Organization

The dissertation is logically organized into three components: background reading, understanding generic libraries, techniques for maintaining generic libraries.

Background reading is presented in Chapter 2, Chapter 3, and Chapter 4. Chapter 2 gives an overview of C++0x templates and the proposed syntax for concepts, and Chapter 3 presents a catalog of programming idioms used in modern C++ libraries and their mapping to the language features in Chapter 2. Chapter 4 describes a validation of the mapping described in Chapter 3. It is important to note that Chapter 2 and Chapter 3 define many of the concepts that underlie the research described in subsequent chapters and should not be overlooked. Chapter 4 is supplemental, but provides insight into the design and implementation of the concepts language features and design techniques for generic library design.

The second component focuses on tools and empirical studies to support comprehension of C++ generic libraries. Chapter 5 describes tools and techniques for reverse engineering and modeling C++0x generic libraries. These tools are applied to
develop a new source code model for template instantiation. Chapter 6 describes an empirical study of generic libraries conducted by extracting programming idiom instances. In Chapter 7, a reference architecture for generic libraries is derived from the study in Chapter 6.

The third component presents research related to the development of tools and techniques for the maintenance of C++ generic libraries. Chapter 8 describes a new technique for identifying concept constraints for function templates, and Chapter 9 discusses how the technique is applied to uncover design flaws in the specification of the iterator concept hierarchy and the assignment of constraints to several algorithms in the C++0x Standard Library.

Conclusions and future work are given in Chapter 10, and Appendix A includes a listing of concept definitions for C++0x for additional reference.
CHAPTER 2

TEMPLATES + CONCEPTS = GENERIC LIBRARIES

This chapter describes the language features (i.e., templates and concepts) that provide the foundation for generic programming and the construction of generic libraries in the C++ programming language. Although templates have been a part of C++ since at least the early 1990’s [Stroustrup 1989; 2007], concepts (as a language feature) are still absent from the language. Despite recent efforts to standardize their syntax [Dos Reis, Stroustrup 2006; Gregor et al. 2006; Siek et al. 2005; Stroustrup 2009b], their addition has been postponed [Stroustrup 2009a] pending further evaluation.

2.1 Templates

The fundamental mechanism for generic programming in C++ is a template. This section presents a brief overview and discussion of language features related to the use of templates to build generic libraries. A more comprehensive treatment of this topic is presented in [Vandevoorde, Josuttis 2002]. Templates are the basic linguistic feature for implementing generic data structures and algorithms in C++, and provide the basis for a number of advanced programming or data structure design techniques such as policy-based class design [Alexandrescu 2001; Czarnacki, Eisenecker 2000] and template metaprogramming [Abrahams, Gurtovoy 2005].

A template is function or class that can be parameterized with user-specified types, integral constants, or even other templates in order to generate (instantiate) new, concrete
functions or classes. These parameterized elements are called template parameters. The template definition is simply a function or class definition that is parameterized by a template.

The substitution for types, values or other templates as arguments for template parameters causes the compiler to instantiate a new class or function definition. This results in new code that is then compiled and optimized for generation. Template instantiation is not a “preprocessor” phase of compilation. In fact, most compilers defer the instantiation of templates until the entire translation unit (the input source file and any other files that it includes) has been processed.

2.1.1 Function Templates

Within the context of generic libraries, function templates can be essentially equated with generic algorithms. These algorithms are written in terms of the generic template parameters. Consider the swap function template in Figure 1.

```
<template<typename T>
void swap(T& x, T& y) {
  T z = x;
  x = y;
  y = z;
}
```

**Figure 1. The swap function template exchanges the values of two objects.**

Here, the template parameter T is the parameterized type. The algorithm swaps the values of the function parameters x and y, both of type T. Calling a function template results in its implicit instantiation. This is to say that no extra work is required on the part of the programmer to cause the compiler to instantiate the template? Writing swap(a, b) will cause the compiler to deduce the types of a and b (say, both as int)
substitute the deduced type (i.e., int) for the template parameter T, and instantiate a new definition for swap. Note that if a and b are not of the same type (e.g., int and double), then the compiler is unable to deduce a viable substitution for T, resulting in a compiler error. This implicitly instantiated implementation of swap over the type int is shown in Figure 2. The result of instantiating a template is called a specialization, although the term template instance is sometimes used interchangeably.

```cpp
template void swap<int>(int& x, int& y) {
    int z = x;
    x = y;
    y = z;
}
```

**Figure 2.** The instantiated swap function with substitution for the type int. The template keyword and swap<int> identifiers indicate a template specialization.

We note that defining a function template implicitly defines the operation or algorithm for a set of types rather than a single type (or set of objects). This can sometimes lead to compiler errors if the user-provided types substituted for template parameters do not define the operators used within the body of the definition. Consider a hypothetical class, File, which is not explicitly copy constructible or assignable (just like the C++ Standard Library stream classes). If we attempt to swap to File objects, the compiler will complain that it could not find a copy constructor or assignment operator, resulting in a compilation error.

In this case, while copying File objects may not be permitted, it is still sometimes useful to swap their underlying file handles (e.g., retargeting logging output). We may, for example, provide the following overload shown in Figure 3.
void swap(File& x, File& y) {
    x.swap(y);
}

Figure 3. A specific version of File swap exchanges underlying file handles without relying on copy or assignment semantics.

In other cases, a data structure may support the required operations, but instantiate the algorithm over those operations would result in (extreme) inefficiencies. In this case, we generally assume that swap is a $O(1)$ (constant) algorithm. However, if we instantiate swap over variable-sized data structure, List<T>, its complexity becomes $O(2m + n)$ with $m$ the number of elements in $x$ and $n$ the number of elements in $y$. This also includes a couple of allocations and deallocations, which can further degrade performance. We can specialize the algorithm for the set of all viable List types by providing the function template overload in Figure 4.

template<typename T>
void swap(List<T>& x, List<T>& y) {
    swap(x->head, y->head);
    swap(x->tail, y->tail);
}

Figure 4. A specialized List swap exchanges only the head and tail pointers of lists rather than copying all of the elements in both lists.

When calling swap over List instances (say, List<int>), the compiler deduces a type assignment to the template parameter T, and instantiates the template using the deduced assignment (i.e., List<int>). C++ allows programmers to explicitly specialize function templates also, but this technique is rarely used in practice and is discussed in this work.

There is one additional technique we can use to specialize function templates. This technique is rooted in the careful use of substitution failure to affect the function
overloading mechanism. Briefly, substitution failure occurs when the substitution of arguments for template parameters results in an expression that causes a lookup failure. Typically, failure implies compilation failure. However, in some contexts (e.g., some phases of function template instantiation), substitution failures do not result in compiler errors. This is referred to as SFINAЕ or Substitution Failure Is Not An Error. SFINAЕ can be applied to remove a function template from consideration as a viable overload. For example, we could write a version of swap that could automatically delegate to a member function defined on the swappable type. This might be written as:

```cpp
template<typename T>
void swap(T& x, T& y, void (T::swap* s)(T&)) {
    ((x).*{s})(y);
}
```

Figure 5. The member swap specialization relies on SFINAЕ to disable the overload for types that do not have a member swap.

This version of the swap algorithm uses SFINAЕ to cause a substitution failure if the type T does not have a member function named swap. Suppose we substitute `int` for T. The declaration and default value third function parameter refer to a nested member function named swap. Since `int` has no nested member functions, the declaration will result in a lookup failure, implying a substitution failure. In this context, the failure is not an error, and the overload is quietly ignored by the compiler. This technique must be used with caution. Without additional safeguards, this particular implementation will cause ambiguous overloads for types that do define a member swap. Specifically, both the fully generic implementation in Figure 1 and the SFINAЕ-constrained implementation in Figure 5 will both be instantiated. If the call is `swap(a, b)`, then
the compiler will not be able to disambiguate the overloads. A systematic method of guarding against these conditions is given in Section 3.8.

An understanding of the C++ overloading mechanics of is critical to understanding some of the more advanced techniques for generic programming and new language features proposed for C++0x. When a function call is resolved (e.g., the call to `swap` in `Error! Reference source not found.`), the compiler constructs a set of candidate overloads. This is constructed by finding viable overloads, instantiating any template declarations, discarding those that result in substitution failures, and the ordering the remaining candidates. The specific rules for partially ordering overloads are complex but are ordered based on type qualifiers, subtype relationships (inheritance), and a tie-breaker between non-template declaration versus template specializations. The specific rules are beyond the scope of this discussion.

2.1.2 Class Templates

A class template is a template whose definition is a class. Class templates are used to build generic data structures and provide support for a number of generic programming idioms, which are discussed in Chapter 3. This section presents an overview of the basic mechanics of defining, specializing, and using class templates.

Consider a basic data structure, `Vector`, a dynamically allocated and re-sizeable array. A generic implementation of `Vector` could be written as shown in Figure 6.
template<typename T>
class Vector {
  typedef size_t size_type;
  typedef T* iterator;
  typedef T const* const_iterator;

  Vector();
  Vector(vector const&);
  template<typename Iter> Vector(Iter f, Iter l);

  void push_back(T const& x);
  void pop_back(T const& x);

  T& operator[](size_type n);
  T const& operator[](size_type n) const;

  bool operator==(Vector const& x) const;
  bool operator!=(Vector const& x) const;

  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;

  T *start, *finish, *extent;
};

Figure 6. An outline of a generic Vector data structure is parameterized over its contained object type T and provides a number of member functions and associated types.

Class templates are essentially identical to non-template classes with the exception that they often contain a number of typedefs that provide names for the types of the template parameters or types associated with the template parameters. Here, the types associated with the Vector are its size_type, iterator and const_iterator. The iterator types are particularly useful since they also abstract the underlying implementation of iterators for the Vector. Here, they are given as pointers, but they could just as easily be defined as wrappers that help enforce valid usage (i.e., safe iterators). Using class templates (i.e., causing their instantiation) is done explicitly. For
example, if we declare a vector of integers as shown in Figure 7 will cause the compiler
to instantiate the Vector template.

```cpp
Vector<int> x;
x.push_back(3);
x.push_back(5);
```

**Figure 7. Declaring and using the Vector causes the compiler to instantiate the template.**

Although this might seem obvious there are some very unintuitive things about this
instantiation. First, simply writing the name `Vector<int>` does not cause the compiler
to instantiate a template because it is not required to be a *complete type*. A type is
complete if the compiler has previously parsed a definition of the object. A type is
required to be complete if the program refers to any of its nested declarations (e.g.,
member functions, constructors, typedefs, etc.). When used with templates, this is
sometimes referred to as *lazy instantiation*. Lazy instantiation is used extensively in
advanced template metaprogramming techniques, such as those discussed in [Abrahams,
Gurtovoy 2005]. Examples of this technique are also given in Chapter 3.

The reason that the Vector template is instantiated in this case is that the declaration
‘Vector<int> x’ requires Vector<int> to be a complete type because the statement
invokes the default constructor. Because a complete type is required for the lookup, the
compiler must instantiate the template.

This leads to the second somewhat unintuitive aspect of the template instantiation
process: member function instantiation is also lazy. Because the program in Figure 7
only refers to the default constructor and `push_back` operations, the compiler will only
instantiate those member functions. No code is generated for any other member
functions. This allows a template to be instantiated piecewise and provides a great deal of flexibility in the definition of the template interface. For example, consider the equality and inequality comparison operators in Figure 7, and suppose that these evaluate equality (or inequality) in terms of the pairwise equivalence of individual terms (via \texttt{operator==} or \texttt{operator!=}). An implementation of \texttt{operator==} is given in Figure 8.

```cpp
Template<typename T>
bool Vector<T>::operator==(Vector const& x) const {
    if(x.size() != y.size()) return false;
    auto i = begin(), j = x.begin();
    while(i != end())
        if(!(*i++ == *j++)) return false;
}
```

\textbf{Figure 8.} The \texttt{Vector} equality comparison operator performs a pairwise equality comparison of equality the elements of \texttt{this} and another \texttt{Vector}.

Here, if the parameterized type \texttt{T} does not define a have an \texttt{operator==}, then the compiler will fail the instantiation of the functions; this results in a compiler error. Note that even the inner comparison uses the \texttt{operator==} and not \texttt{operator!=}. This reduces the requirements on \texttt{T}.

This aspect of template instantiation allows data structure designers to opportunistically define interfaces for more capable underlying data types. In cases where the underlying data types do not provide the required interfaces, the compiler simply fails the instantiation. More advanced generic programming techniques allow generic data structures to “gracefully degrade” their interfaces in terms of time or space performance when viable workarounds are known. Interestingly, this is a well-known
problem in web development, but is under-studied in the domain of generic programming and generic libraries [Florins, Vanderdonckt 2004].

The notion of “degradation” is somewhat similar to the notion of class template specialization. In C++, a user can specify an alternative definition for a class template based on the kinds of types being substituted for template parameters. Class template specialization is often used to achieve time or space optimizations or when the template is expected to accept a set of (possibly) unrelated and statically non-polymorphic types.

For example, if a programmer wants to create a Vector of boolean values, it is wasteful to represent each boolean value as its base integral type, which is usually the size of char or int. A more space-efficient solution would encode each boolean value as bit in a bit-string. This can be accomplished using explicit class template specialization as shown in Figure 9.

```cpp
template<>
class Vector<bool> {
    // ... similar vector interface
    word_type *start, *extent;
    size_type finish;
};
```

**Figure 9.** An outline of a Vector specialization for boolean values encodes each value as a bit rather than an integral representation.

The specialization is explicit because it automatically provides concrete type arguments (bool) for the template parameter T in the generic vector definition given in Figure 6. The set of specialized arguments are given following the class name. The sequence of arguments over which the template is specialized must match the sequence of template parameters given in the generic definition. Because this class is defined in terms of concrete types, it is not actually a template—even though it is preceded by the
template<> tokens. When specialized the “fully generic” or non-specialized Vector is called the primary template.

A class template may also be partially specialized. A partial class template specialization is a class template specialization that is a) a template and b) specialized over the names of template whose arguments refer to the specializations template parameters. Said otherwise, a partial template specialization is a specialization whose specialized arguments are given as other class templates. For example, we can generalize the notion of a compact integral encoding using the Compact_Value type in Figure 10.

```cpp
template<typename T, size_t N>
class Compact_Value {
  typedef T value_type;
  static constexpr size_t width = N;
};
```

**Figure 10.** The Compact_Value class encodes information about a type T and its minimum bitwise encoding width N.

Note that the constexpr declaration specifier is new in C++0x and indicates that the declared variable (or function) can be evaluated at compile time. This simple class template can be used to partially specialize the Vector class template in order to create a compact-encoded vector for integral types. This partial template specialization is shown in Figure 11.

```cpp
template<typename T, size_t N>
class Vector<Compact_Value<T, N>> { 
  // vector interface
  word_type *start, *extent;
  size_type finish;
};
```

**Figure 11.** A partial template specialization of the Vector over the Compact_Value class template can be used to implement space-optimal vectors for integral types T with minimum bit encodings of width N.
The arguments over which the template is specialized appear similarly to the `Vector<bool>` specialization, but refer to the `Compact_Value` template over the type `T` and the unsigned integer `N`. When instantiating a `Vector` template, the compiler matches the given arguments to a template specialization who’s pattern exactly matches the template arguments. If no such pattern can be, the primary template is instantiated instead. Not that instantiating a partial template specialization requires the “inner” template arguments to be deduced, just as with implicit function template instantiation.

### 2.1.3 Non-Type and Variadic Template Parameters

As demonstrated in Figure 10 and Figure 11, C++ also supports template parameters that are not specified as type names. Specifically, templates can be parameterized over *non-type template parameters* and *template template parameters*. Additionally, C++0x introduces a new feature called variadic templates, which allows a template to be defined in terms of an arbitrary number (but the same kind) of template parameters. These features are introduced here since they are discussed throughout the context of this work.

Non-type template parameters include integral constants and the (extremely) rare pointer/reference template parameters. Integral constant template parameters provide a method of varying a constant value of a data structure between its different instantiations. Although examples have been given in Figure 10 and Figure 11, the best known example is a wrapper for a stack-allocated, constant-sized array, which is shown in Figure 12.
template<typename T, size_t N>
class Array {
    Array();
    Array(Array const& x);
    T arr[N];
};

Figure 12. The Array wrapper provides uniform C++ semantics (e.g., copyability) for C arrays. It is parameterized over a type and size.

Here, the parameter size_t (typically an alias for unsigned int) is a non-type or integral constant template parameter. It can be specified, for example, by writing the type name Array<int, 10> to denote a stack-allocated array of 10 integers.

An interesting side effect of using integral template parameters is that the value is “projected” into the type system. For example, if we define a template named Number that takes an int template parameter, the specialization Number<0> will be a different type than Number<1>. Unless some conversion functions are implemented, these two types are entirely unrelated.

As mentioned, there are rare cases in which a template parameter may be written as a pointer or reference type, whose arguments are constant. For example, consider the class template definition in Figure 13.


```cpp
template<typename T, T Func(T)>
struct Unary_Function {
  typedef result_type T;
  typedef argument_type T;
  Unary_Function(F* f) : f(f) {
  }
  T operator()(T x) const {
    return f(t);
  }
  F* f;
};
```

**Figure 13.** The `Transform` class template defines a wrapper functor for all unary functions on a type `T`.

The template can be instantiated by writing `Unary_Function<float, sin>`, for example. Although the use of such a facility is not immediately obvious, it does provide one very interesting feature: it projects function pointers into the type system. This allows us to reason about different functions with the same type even though they point to different functions. In a sense, this feature of projecting compile-time information into the type system is weakly analogous to reflection, largely due to the class template specialization capabilities of C++. While this feature does not give the programmer the ability to directly query objects for their types and properties, it can be used to build complex *metaprograms* that reason about type arguments. These concepts are explored more thoroughly in Chapter 3.

In addition to non-type template parameters, C++ allows us to specify template template parameters—template parameters whose values are other templates. These parameters provide another mechanism for lazy template instantiation (although less powerful than other techniques). Consider the definition of the `Stack` adaptor in Figure 14.
template<typename T, template<typename> Container = List>
class Stack {
public:
    T& top();
    T const& top() const;
    void push(T const&);
    void pop();
private:
    Container<T> data;
};

Figure 14. The Stack adaptor is parameterized over a type T and the name of a container data structure, which defaults to the List template.

We describe Stack as an adaptor because it adapts the interface of an underlying container to that of a stack abstract data type. Here, the underlying container is given as a template template parameter, named Container and defaulting to List. The parameter is specified in terms of its “template signature”—it’s arity and kinds of parameters it accepts, here a single type name. The Stack implementation instantiates the Container template over the type parameter T in order to compose a new data type.

A variadic template parameter is a template parameter specification that can accept any number of arguments (i.e., zero or more). One use of variadic template parameters is the ability to create type-safe variadic function templates. This can be used in conjunction with the C++0x rvalue reference and forwarding function to create so-called “forwarding constructors”. A forwarding constructor allows a generic data structure to forward an arbitrary number of arguments through a function call to an underlying data type. Consider the implementation of a List_Node class.
template<typename T>
struct List_Node {
    template<typename... Args>
    List_Node(List_Node* p, List_Node* n, Args&&... args)
        : prev(p), next(n), data(forward<Args>(args))
    
    T data;
};

Figure 15. The List_Node class uses forwarding constructor to construct its link structure and underlying data object.

The template constructor definition specifies a variadic type parameter, Args, which is also called a parameter pack. A template that takes a variadic template parameter is also called a variadic template. The ellipses following a template parameter declaration denotes that the declared parameter will be a parameter pack. The last function parameter, Args&&... args, uses the pack expansion operator to expand the parameter pack into a (possibly empty) sequence of comma-separated function parameter declarations, each being passed by rvalue-reference. The initialization of the data member is constructed over the expansion of the packed expression 'forward<Args>(args)'. In essence, this constructing a function call over the argument types and values specified in the packed arguments. A compiler error results if the List_Node’s data member (of type T) does not have a constructor that matches the call. If Args is an empty parameter pack, then the constructor call will try to match the default constructor. If Args contains a value whose type is T const&, then the constructor call will try to match T’s copy constructor. For example the implementation of push_back for the List data structure in
template<typename T>
List<T>::push_back(T const& x) {
    List_Node<T>* p = new List_Node<T>(forward<T const&>(x));
    // Connect p to the list
}

Figure 16. The construction of a new List_Node forwards a copy of the given object to the node’s constructor, which is in turn forwarded to T’s copy constructor.

Variadic template parameters can also be used with class templates, and have an interesting affect on the definition of partial class template specializations. Consider the definition of a Tuple data structure—a heterogeneously typed, compound data type—in Figure 17

template<typename... Args> class Tuple;

template<>
class Tuple<>
{
};

template<typename T, typename... Rest>
class Tuple<T, Rest...>
: Single<T>, Tuple<Rest...>
{
...};

Figure 17. The Tuple data type is a recursively defined variadic template with two primary specializations. The specific implementation is omitted.

The first declaration of Tuple indicates that it is a variadic template with no definition. Essentially, this indicates that the template’s implementation will be covered by one or more partial template specializations. The second definition of Tuple matches the case where the Args parameter pack is empty. The third and most interesting definition is given recursively. This partial specialization matches the case where there is at least one type parameter followed by zero or more additional type parameters. Its implementation is accomplished by inheriting first from a wrapper for the single matched
type and second, by a recursive expansion of the remaining type arguments (which could be empty).

The specialization of variadic template parameters adds another layer of depth to the template specialization structure. Specifically, specializations of a variadic template parameter are primary templates that can be further (explicitly or partially) specified by concrete types or other templates. For example, we could further define a partial specialization of `Tuple` that can be used to flatten nested tuple declarations as shown in Figure 18.

```
template<typename... Args, typename... Rest>
class Tuple<Tuple<Args...>, Rest...>
  { ... };
```

**Figure 18.** A partial template specialization of `Tuple` that causes the data structure to be self-flattening.

With this specialization, an instantiation of the form `Tuple<Tuple<T>>` is structurally but not nominally equivalent to `Tuple<T>`.

Variadic templates are not limited to type parameters. We can also create parameter packs for non-type template parameters and template template parameters. For example, the declarations in Figure 19 are valid class template definitions.
template<typename T, size_t... Dims>
class Vector_Field { };

template<typename T, T... Values>
void sum(Values... args);

template<typename T, T (Funcs...) (T)>
void call(T (Funcs* func)(T) ...);

template<template<typename> class... Args>
struct Unary_Templates { };

template<template<typename...> class... Args>
struct Any_Templates { };

Figure 19. Declarations of variadic non-type and template template parameters.

The Vector_Field class is uses variadic integral parameters to describe an n-Dimensional vector field of type T. The variadic sum function computes the sum of a number of integral values. The call function can be used to delegate invoke a sequence of functions. The Unary_Templates class can be parameterized over a sequence of templates, all of which take exactly one type parameter. Likewise, the Any_Templates class can be parameterized over a sequence of templates that take any number of type parameters.

Despite all of the mechanisms that can be used to define, abstract, and specialize generic components, the C++ programming language still omits an essential feature: the ability to describe constraints on template parameters. In the current version of C++, it is possible to substitute any type for a (type) template parameter even though it may result in a compilation error, and when template usage is extensive these errors can become quite verbose and difficult to understand. The proposed solution to this (and other) shortcomings is a language feature called concepts.
2.2 Concepts

Concepts provide a mechanism for describing the abstractions represented by template parameters in C++. By describing the abstraction, we can effectively constrain the set of types (or other “values” of template parameters) over which a template can be instantiated. As a language feature, concepts are rooted in the need to algebraically or axiomatically describe requirements and properties of abstract data types [Guttag 1977] in order to support (semi-)automated program verification [Guttag, Ellis, Musser 1978].

These notions were further developed for parameterized programming [Goguen 1984] as an alternative to functional based languages [Denhert, Stepanov 1998]. This approach favors the top-down program composition over bottom-up type inference and algebraic reasoning. The Tecton language [Kapur, Musser, Stepanov 1981] introduced the term “concept” as a means of specifying the syntactic and semantic requirements of types in a generic program, which could also be used to constrain type or program arguments in a compositional framework. One application of this language was the semi-automated validation of programs [Kapur, Musser, Nie 1992].

The top-down approach to program composition supports iterative development and refinement of generic components. These techniques were applied to define generic libraries for common computing utilities and their associated concepts. This work ultimately resulting in the development of the Standard Template Library [Musser, Stepanov 1988; 1994]. Currently, the specification of concepts for STL are given as supplemental documentation [Austern 1998].
Efforts were made to harmonize competing proposals for C++ concepts, from Texas A&M [Dos Reis, Stroustrup 2006] and the University of Indiana [Garcia et al. 2003; 2007; Siek et al. 2005; Siek, Lumsdaine 2005a; b], and standardize them as a part of the C++0x programming language [Gregor et al. 2006], but the addition was ultimately postponed. In this section, we present the version of concepts that was ultimately removed from the C++ specification. A brief overview the Texas A&M proposal is also given for perspective.

2.2.1 Concept Definitions

In C++, a concept defines a set of requirements on a type or set of types indicated by a sequence of template parameters. More precisely, a concept can be thought of as a predicate on template parameters that determines whether or not the substituted template arguments satisfy all of the requirements expressed within the concept body. Consider two basic concept definitions for iterator types shown in Figure 20.
concept Iterator<typename X>
   : Semiregular<X>, Equality_Comparable<X>
{
   Signed_Integer difference_type = X::difference_type;
   Move_Constructible reference = X::reference;

   reference operator*(X&&);
   X& operator++(X&);
   X operator++(X& x, int) {
      X t(x); ++x; return t;
   }
}

concept Input_Iterator<typename X>
   : Iterator<X>
{
   Object_Type value_type = X::value_type;
   requires Convertible<reference, value_type const&>;
}

Figure 20. Concept definitions for Iterator and Input_Iterator types express required interfaces and behaviors.

The Iterator concept expresses the basic requirements for all iterators, which minimally specifies the ability to move and dereference them. Very few behavioral requirements are given for such a broadly specified category of types. Specifically, the Iterator concept defines a predicate on a single type parameter X. The concept can be “queried” by writing it as an explicit template instantiation over some type T as Iterator<T>. If the predicate is satisfied, we say that T is a model of Iterator, that T models Iterator, or that T is an Iterator. We might also say that Iterator<T> is a valid model. These are all equivalent statements indicating the satisfaction of the predicate.

This body of this concept defines two associated types: difference_type and reference. It is important to note that these type names are defined within the scope of the concept and are not directly associated with the template parameter X. Here, these
type names are assigned default types. These two types are also constrained using a shorthand constraint notation. Specifically, *difference_type* is required to be a kind of *Signed_Integer*, and *reference* must be *Move_Constructible*.

The concept declares a set of *associated functions*, here *operator* and *operator++* (both pre- and post-increment) that are required to be implemented by or overloaded for the parameter *x*. A canonical definition of postincrement is given in terms of the preincrement, allowing types to omit the definition. The predicate is valid if and only if a) concrete type names can be assigned to the associated types and b) overloads (built-in or user-defined) can be found for the specified associated functions.

The *Iterator* concept *refines* the *Semiregular* and *Equality_Comparable* concepts. Concept refinement is roughly analogous to class inheritance, where requirements and declarations in the general concept (e.g., *Semiregular*) are aggregated within the refined concept (i.e., *Iterator*). This is to say that the refining concept inherits the associated type names and any defined operations from the refined concept. Here, *Iterator* is “inheriting” requirements on *X* for copy and move constructability and equality and inequality comparison. We note that a refined concept represents a subset of types that are represented by a general concept.

The *Input_Iterator* concept also *refines* the *Iterator* concept, inheriting the syntactic requirements common to all iterator types, and adding new *associated requirements*. An associated or aggregate requirement is an expression of requirements, typically on associated types. Here, *Input_Iterator* requires the reference type to be *Convertible* to the *value_type*. Technically, the constraints on *Iterator’s*
difference_type and reference are also associated requirements. Concept definitions are sometimes organized as hierarchies in order to define a taxonomy of types.

A conceptual outline of the C++ iterator hierarchy is given in Figure 21.

```cpp
concept Forward_Iterator<
type X>:
   Input_Iterator<X>
{
   axiom Multipass(X a, X b) {
      (a == b) => (++a == ++b) && ((void)++X(a), *a) == *a;
   }
}

concept Bidirectional_Iterator<
type X>:
   Forward_Iterator
{
   typename postdec_result;
   X& operator--(X&);
   X operator--(X& x, int) {
      X t(x); --x; return t;
   }
}

concept Random_Access_Iterator<
type X>:
   Bidirectional_Iterator<X>
{
   X& X::operator+=(X&, difference_type);
   X operator+(X const& x, difference_type n) {
      X t(x); t += n; return t;
   }
   X& X::operator-=(X&, difference_type)
   X operator-=(X& x, difference_type) {
      X t(x); t -= n; return t;
   }
   difference_type operator-=(X const& x, X const&);
   reference operator[](X&& x, difference_type n) {
      return *(x + n);
   }
}
```

**Figure 21.** The concept hierarchy defines a taxonomy of iterators based on their supported operations and semantics.

This concept hierarchy defines several kinds of iterators, largely differentiated by their supported operations. The Forward_Iterator, however, is differentiated from
the base-most `Iterator` concept by a *semantic requirement* or *axiom*. An axiom is a statement about the behavior of types modeling the concept. Here, the axiom states that a type `X` is a `Forward_Iterator` if and only if a) if two iterators refer to the same object, then they will refer to the same object after incrementing both, and b) incrementing the iterator does not cause the referenced object to be destroyed (i.e., an iterator reference is valid after an equivalent iterator is incremented). It is important to note that, unlike associated types and functions, axioms cannot be statically verified by the compiler.

A *concept map* provides a means of explicitly adapting a type (or set of types) to a concept definition. For example, pointers are valid models of random access iterators, but fail to meet the requirements of the `Random_Access_Iterator` concept definition because no associated types (e.g., `value_type`) can be automatically deduced. As such, we explicitly map pointer types onto the concept, telling the compiler that we should use this concept map as the model instead of the concept definition. An example is shown in Figure 22.

```cpp
template<typename T>
class concept_map Random_Access_Iterator<T*> {  
  typedef T value_type;  
  typedef ptrdiff_t difference_type;  
  typedef T& reference;  
  // ...  
};
```

**Figure 22.** A concept map adapts pointer types to the `Random_Access_Iterator` concept.

A reference to the model `Random_Access_Iterator<int*>` will refer to the concept map rather than the concept. Note that since `Random_Access_Iterator` is a refinement of other iterator concepts, this concept map will also satisfy the requirements
of the refined concepts. Said otherwise, If Random Access Iterator<int*> is a valid model, then so is Iterator<int*>.

2.2.2 Constrained Templates

The primary role of concepts is to constrain template parameters by limiting the set of types that can be substituted for those parameters to only those that are valid models of the concept. Consider the find algorithm pictured in Figure 23. Here, the requires clause gives a conjunction of constraints on model. Specifically, each model must be valid in order for the template to be instantiated. The process of enforcing requirements and constraints is called concept checking. If the concept check fails, then a compiler error is emitted at the point of failure.

```c++
template<typename Iter, typename T>
requires InputIterator<Iter>
    && HasEqual<
        typename InputIterator<Iter>::value_type, T
    >
Iter find(Iter b, Iter e, T const& x) {  
    for( ; b != e; ++b) if(*b == x) return b;  
    return e;  
}
```

**Figure 23.** The find algorithm requires that the Iter template parameter satisfy the requirements of Input Iterator and that an operator== overload exists for the iterator’s value_type and T.

Concept checking enables the generation of substantially clearer error messages than without. For example, suppose a programmer unintentionally causes Iter to be substituted with int. Obviously, the int type does not model the Input Iterator concept because no operator* is defined that takes int as its argument type. Since the error is caught by the concept check, the resulting message can be generated with respect
to the concept rather than the failed expression lookup, which might appear as, “no
operator* in the expression *b == x”.

The interplay between constrained function templates and the function overloading
mechanism is straightforward. Consider the distance algorithm overloads show in
Figure 24. Note the use of shorthand constraint notation, in which the concept name
replaces the typename declaration specifier in the template parameter list.

```cpp
template<typename Iter>
type distance(Iter b, Iter e) {
  type n = 0;
  for(; b != e; ++ b, ++n) ;
  return n;
}

template<typename Random_Access_Iter>
type distance(Iter b, Iter e) {
  return e - b;
}
```

**Figure 24.** The distance operation is overloaded on the **Iterator** and
**Random_Access_Iterator** concepts, allowing the compiler to differentiate
constant and linear time implementations based on the kind of iterators.

Without constraints, the second distance overload would appear to be a
redeclaration of the first, which will result in a compiler error. However, a constrained
overload is allowed to have the same signature, but vary in constraints.

Suppose we try to call the distance function by substituting int as a template
argument for Iter, which we expect to fail both constraints since int is not an
**Iterator** and also (transitively) not a **Random_Access_Iterator**. Failing a
constraint triggers a substitution failure, causing the overload candidate to be removed
from the candidate set. Substituting int for Iter will result in an error message stating
that no overload can be found (and hopefully emit diagnostics for the failed overloads).
If the substituted template argument is `List<T>::iterator` (i.e., it models `Iterator` but not `Random_Access_Iterator`), then the second overload will be removed from consideration due to the constraint (substitution) failure. If the substituted template argument is `Vector<T>::iterator` (i.e., a pointer and therefore a model of the `Random_Access_Iterator` concept), then both function templates are considered viable overloads. In this case, the compiler must partially order the overloads with respect to the most refined concepts of each constrained template parameter. Since pointers are `Random_Access_Iterators`, the second overload is chosen over the first. Although not shown, concept requirements can also be negated. For example, it is possible to write a requirement for `!Iterator<T>`, which will be true if `T` does not model the iterator concept.

Class templates and member functions can also be constrained using concepts. Figure 25 shows a partial listing of the `Pair` class template and its constraints. Here, the template parameters `T` and `U` are required to model the `Variable_Type` concept, implying that `T` and `U` can be any type for which the programmer can declare a variable, which is fairly unrestrictive.

```cpp
template<typename T, typename U>
    requires Variable_Type<T> && Variable_Type<U>
struct Pair {
    requires Less_than_Comparable<T> && Less_than_Comparable<U>
    bool operator<(Pair const& x) const;
};
```

Figure 25. A partial listing of the `Pair` class template and its constraints. Template parameters `T` and `U` are required to model the `Variable_Type` concept.
The `operator<` member function further constrains the template parameters `T` and `U` to be `Less_Than_Comparable`. Recall that member functions are only instantiated when used. If the programmer calls `Pair<T, U>::operator<`, then compiler will have to validate the `Less_Than_Comparable` constraints—but only if the program includes that specific call. Here, it would be inappropriate to enforce those constraints on the `Pair` as a whole, since it would be overly restrictive.

### 2.2.3 Concept Checking

There are two distinct approaches to determining how concepts are evaluated or checked: automatically or explicitly. There are advantages and disadvantages to both approaches, especially for algorithm and type providers. The difference between automatic and explicit concepts is obvious when considering the difference between the `Input_Iterator` and `Forward_Iterator` concepts in Figure 20 and Figure 21.

The `Input_Iterator` concept can be checked automatically. This is to say that the compiler can automatically evaluate all of the requirements of the concept by performing basic name and operator lookup operations. If a lookup fails, then concept is not valid. The `Forward_Iterator` concept cannot be checked automatically because axioms cannot be statically validated by the compiler. Since the `Multipass` axiom cannot be automatically checked, the `Forward_Iterator` concept is not syntactically differentiable from the `Iterator` concept.

*Syntactic differentiation* is the idea that a semantic action corresponds to a single syntactic interface. In this case, `Input_Iterators` (e.g., input stream iterators) could
be syntactically differentiated from Forward_Iterators by differentiating the semantics of copy construction. This is to say that both kinds of iterators are copy constructible, but multiple of Input_Iterators on the same sequence are not independent on each other. Dereferencing one iterator may invalidate all other copies, potentially resulting in undefined behavior if dereferenced. Providing an interface for explicit state-saving copies will differentiate the copy/share semantics of input and forward iterators [Alexandrescu 2009]. However, this has not been done for the C++ Standard Library. This problem is also highlighted by the Vector’s range constructors shown in Figure 26.

```
template<typename T>
class Vector {
    template<Iterator Iter>
    Vector(Iter f, Iter l) { // One overload
        while(f != l) push_back(*f++);
    }

    template<Forward_Iterator Iter>
    Vector(Iter f, Iter l) { // Explicit overload
        resize(distance(f, l));
        copy(f, l, begin());
    }
};
```

**Figure 26.** Two range constructors for the Vector data type are differentiated by the kinds of Iterators.

When instantiating the range constructor, both overloads accept the same sets of Iterator types, causing ambiguous lookups. One solution is to make the Forward_Iterator concept explicit by prefixing the concept declaration with the keyword explicit. Explicit concepts inhibit automatic checking by requiring the programmer to provide concept maps.
Another language feature described in [Stroustrup 2009b] describes a feature called “explicit refinement.” This is a way of allowing automatic concepts to explicitly satisfy explicit concepts. To demonstrate this idea, consider definitions of various concepts for the C++0x type system in Figure 27.

```cpp
concept Object_Type<typename T> {
    requires True<is_object<T>::value>;
}

explicit concept Class_Type<typename T> : Object_Type<T> {
}

concept Class<typename T> : explicit Object_Type<T> {
    requires True<is_class<T>::value>;
}

concept Union<typename T> : explicit Object_Type<T> {
    requires True<is_union<T>::value; 
}
```

**Figure 27. A classification of types different basic Object_TYPES from Classes and Unions.**

In this example, the Class_Type concept is not differentiable from Object_Type. If checked automatically, then every Object_Type would also be classified as Class_Types, which simply is not the case. As such, the concept is declared explicit. However, this would place the burden on either the compiler (or worse) the user to explicitly cause every class and union to be mapped to the Class_Type concept. Instead, we can use *explicit refinement* to allow a satisfied refinement to explicitly denote the satisfaction of the refined class. Here, the Class and Union concepts explicitly refine the Class_Type concept.
2.2.4 An Alternative Syntax

The language features given above are largely derived from the Indiana University proposal [Siek et al. 2005]. The approach taken in the Texas A&M proposal [Dos Reis, Stroustrup 2006] is somewhat different. Rather than express associated function requirements in terms function signatures, operational requirements are given as valid expressions. For example, consider the Iterator concept shown in Figure 28.

```cpp
concept Iterator<typename X> {  
typename X::difference_type;
typename X::reference;

Var<X> i;
X& pre = ++i;
X const& post = i++;
reference r2 = *i;
}
```

Figure 28. The Iterator concept written using valid expressions.

The concept shown in Figure 20 that uses function signatures and the concept in Figure 28 are equivalent specifications. However, this approach to the specification of concepts does not provide the adaptive features of the former. There is currently no way to provide default semantics for the required operations, nor does the alternative syntax support refinement. It is worth noting that refinement is also excluded from the specification of concepts in [Stepanov, McJones 2009]. There is also a difference in how associated types are defined, and this is effectively demonstrated by looking at the concept mapping for pointers to Random_Access_Iterators.
template<typename T>
assert template<typename T> Random_Access_Iterator<T*> with {
    T::*value_type = T;
    T::*difference_type = ptrdiff_t;
    T::*reference = T&;
};

Figure 29. The concept assertion is roughly analogous to a concept map, and explicitly adapts a type to a concept.

Although roughly analogous to concept maps, the concept assertion could be seen as “injecting” the required type names into its adapted type. This allows algorithms to be written in terms of types associated with the template parameter rather than the concept.

One final difference is the specification of constraints. In the Texas A&M proposal a constraint (the requires clause) simply evaluates a boolean constant expression. This implies that writing an identifier of the form Iterator<T> is actually an alias for the boolean constants true or false. The context in which such identifiers are written determines how the compiler applies the results.

2.3 Generic Libraries

The style of programming engendered by these language features is at root of a large (and still increasing) number of generic libraries. The Standard Template Library (STL) is the archetypal example of a generic library [Austern 1998] for C++ and helped lay the foundations for every C++ generic library that followed. Building on techniques and ideas pioneered in Ada [Musser, Stepanov 1987; 1988; 1994], the STL has been adopted as a part of the C++ Standard Library and is arguably one of the most widely used libraries ever used. Beyond that, there are many generic libraries targeting a wide variety of domains including: graph data structures and algorithms [Kühl 1998; Siek, Lee,

A number of different styles of generic programming have begun to emerge within this domain. Many of the previously mentioned libraries follow the basic precepts of generic programming as set forth by Stepanov and Musser [Musser, Stepanov 1988; Stepanov, McJones 2009]. However, there have also been innovations in the use of generic programming for generative programming and policy-based class design [Alexandrescu 2001; Czarnecki, Eisenecker 2000], template metaprogramming [Abrahams, Gurtovoy 2005], expression templates [Veldhuizen 1995], and domain-specific embedded libraries [Prud'homme 2006].

All of these libraries and the techniques used in their design and implementation are rooted in templates and the underpinnings of concepts. However, because concepts are not yet part of the C++ programming language, none of these libraries can take advantage of the features offered. The absence of language features has resulted in the development of programming idioms to emulate and encapsulate some of those features. Additionally, there have been several different approaches to codify some of the features offered by concepts [Siek, Lumsdaine 2000; Zólyomi, Porkoláb 2004], but these have not been widely adopted, in some cases, perhaps, because of the intrusiveness of their usage.
CHAPTER 3

IDIOMS FOR GENERIC PROGRAMMING

In this chapter, we describe the programming idioms used in the creation of C++ generic libraries. The contribution of this chapter is to present a mapping between modern techniques for generic programming and the language features proposed for concepts described in Chapter 3. This mapping is presented in Table 1.

Table 1. A mapping of C++0x concept features to programming idioms used in modern C++ generic libraries

<table>
<thead>
<tr>
<th>Programming Idiom</th>
<th>Concept Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type traits</td>
<td>Concepts as predicates, abstractions</td>
</tr>
<tr>
<td>Traits classes</td>
<td>Concepts as abstractions</td>
</tr>
<tr>
<td>Traits class specializations</td>
<td>Concept maps</td>
</tr>
<tr>
<td>Tag classes</td>
<td>Concepts as properties</td>
</tr>
<tr>
<td>Tag class hierarchies</td>
<td>Concept hierarchies</td>
</tr>
<tr>
<td>Concept-controlled polymorphism</td>
<td>Constrained templates</td>
</tr>
<tr>
<td>Tag dispatch</td>
<td>Concept-based overloading</td>
</tr>
<tr>
<td>Extended substitution failure</td>
<td>Automatic concept checking</td>
</tr>
</tbody>
</table>

The features of concepts proposed for the C++0x language address several different aspects of generic programming in modern C++. The concept element assumes responsibilities for acting as a predicate, supporting type abstraction, and supporting properties of types. These different responsibilities are addressed by different (although related) programming idioms in the current language. In this chapter, we present a number of idioms used in the definition and implementation of C++ generic libraries. In
cases where the idiom is addressed by a C++0x language feature, we describe how each idiom is related or motivated the new language feature.

A number of techniques related to this style of programming have been described in research or industry literature. Coplien provides the first description of common template patterns in generic components [Coplien 1995]. Specific design patterns for generic programming are presented in [Duret-Lutz, Géraud, Demaille 2001], and generative techniques for datatype composition (i.e., mixins) are described in [Smaragdakis, Batory 2001] and [Czarnecki, Eisenecker 2000]. Template metaprogramming and its associated idioms/patterns are described by Abrahams and Gurtovoy [Abrahams, Gurtovoy 2005]. Alexandrescu employs a number of these techniques in his treatment of C++ library design [Alexandrescu 2001]. A technique for implementing concept-controlled polymorphism is presented in [Järvi, Willcock, Lumsdaine 2003].

3.1 Functors

A functor (also 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 encapsulating a function as an object, functors be used to a) abstract or adapt calls to other functions, b) curry function parameters, or c) maintain state (including external data references) between calls. Although functors are more traditionally associated with functional programming, their manifestation in C++ is heavily dependent upon the language features for generic programming. An example of a functor is given in Figure 30.
template<typename T>
struct greater {
    bool operator()(T x, T y) const {
        return x > y;
    }
};

Figure 30. The \texttt{greater} functor encapsulates the comparison, via \texttt{operator\textgreater}, for a type \texttt{T}.

Here, the class template \texttt{greater} is a functor that abstracts the syntactic comparison of objects via \texttt{operator\textgreater}. This can be used to parameterize high order functions or algorithms such as \texttt{sort}. For example, we could define a high-order function, \texttt{is\_sorted} that determines whether a sequence of objects obeys an ordering determined by a binary relation (less than, greater than, etc.). Consider its definition in Figure 31.

```
template<typename Iter, typename Comp>
bool is_sorted(Iter f, Iter l, Comp cmp) {
    Iter i = next(f);
    for( ; i != l; ++i, ++f) {
        if(!cmp(*f, *i)) return false;
    }
    return true;
}
```

```
int a = {5, 4, 3, 2, 1};
assert(is_sorted(a, a + 5, greater<int>()));
```

Figure 31. The \texttt{is\_sorted} function is a high-order function parameterized over a binary relation, \texttt{Comp}.

Functors are the basis of C++0x lambda expressions. A lambda expression defines an anonymous functor within the scope in which it is written. For example, we could opt to call \texttt{is\_sorted} via an equivalent lambda function as shown in Figure 32.

```
int a = {5, 4, 3, 2, 1};
assert(is_sorted(a, a + 5, [] (int x, int y) { return x > y; }));
```

Figure 32. The \texttt{is\_sorted} algorithm is called using a lambda expression to define and instantiate an anonymous functor.
Although C++0x lambda expressions reduce the programmer’s need for creating functors, they do not fully deprecate the idiom. We suspect that named functors will continue to be useful in generic libraries.

3.2 Type Traits

Type traits are compile-time mechanisms for evaluating properties of types or associating one type with another. Type traits are typically implemented as template metafunctions. A template metafunction is a class template that contains a nested type declaration (or typedef) or integral constant expression, typically named `type` or `value`, respectively. The nested type or value acts as the “return value” of the metafunction, and metafunction evaluation is done by instantiating the template.

A metafunction containing a static constant value is sometimes referred to as a `type predicate`. For example, consider the `is_same` type trait shown in Figure 33.

```cpp
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;
};
```

**Figure 33.** The `is_same` type trait returns true if the two types are the same and false otherwise.

The primary template `is_same` type trait return `false` for any two types. Only in the case where the two types are the identical, will the trait return `true`. The “true” case is implemented as a partial class template specialization of the primary. Although this
trait is very simple, other type traits can be quite complex (e.g., `is_function`), and others may require compiler support (e.g., `is_polymorphic`).

A type trait that associates one type with another (as either a mapping or computed type) is sometimes referred to as a type accessor. For example, the `result_of` type trait in Figure 34 yields the return type of a function type.

```cpp
template<typename F>
struct result_of {
    typedef F::result_type type;
};
template<typename R, typename... Args>
struct result_of<R(Args...)> {
    typedef R type;
};
```

**Figure 34.** The `result_of` type trait allows programmers to access the result a function type or functor.

As with the `is_same` type trait depicted in Figure 33, the `result_of` type trait is implemented as a primary and specialized class template. Here, the primary template is instantiated for any type that is not a function type. The specialization is a variadic template that matches function types returning the type R and taking a sequence of type arguments.

With respect to concepts, type predicates can be used to determine if some property of a type holds. This is analogous to the use of concepts as a method of writing predicates on a type and aligns well with the Texas A&M notion of evaluating concepts as predicates [Dos Reis, Stroustrup 2006]. Type accessors are somewhat analogous to the use of associated types in concept definitions in the Indian University proposal [Gregor et al. 2006; Siek et al. 2005]. They provide a mechanism of accessing associated or derived type information without having to specify it directly on the argument type.
3.3 Traits Classes

A traits class is a class template that provides a mechanism for decoupling types associated with a data structure from its implementation. Traits classes are (partially) specialized to adapt concrete types (or sets of types) to the generic abstraction specified by the primary template. The iterator_traits class from the C++ Standard Library in Figure 35 is a canonical example.

```cpp
template<typename Iter>
struct iterator_traits {
    typedef typename Iter::value_type value_type;
    typedef typename Iter::difference_type difference_type;
    typedef typename Iter::reference reference;
};

template<typename T>
struct iterator_traits<T*> {
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T& reference;
};
```

Figure 35. The iterator_traits class decouples the types associated with iterators from their implementation. The partial specialization supplies the requisite type names for pointer types.

The iterator_traits class provides a kind of static façade for the type names associated with the iterator abstraction. This is crucial when valid iterator types (esp., pointers) cannot provide nested types, or the type being adapted to the pointer abstraction cannot be modified. In this case, the traits class can be specialized to provide those associated type names. With this idiom in place, the identifier iterator_traits<I>::value_type will correctly name the value type associated with an iterator—even if the iterator is a pointer.
Like type accessors, traits classes are roughly analogous to the declaration of associated types within a concept definition. There is little difference between the concept’s type name `Iterator<Iter>::reference` and the traits class type name `iterator_traits<Iter>::reference` except the mechanisms used to define them. Typically traits classes are preferred over type accessors when used to represent an abstraction rather deriving a single type.

### 3.4 Tag Classes and Hierarchies

A *tag class* is typically an empty non-template class that symbolically denotes a compile-time property of a type. Tag classes are often defined in inheritance hierarchies called a *tag hierarchy*. The tags and accessor that describe iterators in the C++ Standard Library are summarized in Figure 36.

```cpp
struct input_iterator_tag { }
struct forward_iterator_tag : iterator_tag { }
struct bidirectional_iterator_tag
    : bidirectional_iterator_tag { }
struct random_access_iterator_tag
    : bidirectional_iterator_tag { }

template<typename T>
struct iterator_category {
    typedef typename T::category type;
};

template<typename T>
struct iterator_category<T*> {
    typedef random_access_iterator_tag type;
};
```

*Figure 36. A tag class hierarchy defines a taxonomy of iterator kinds, and a type accessor provides a method of determining an iterator’s kind.*
This tag class hierarchy parallels the iterator concept hierarchy given in Figure 21. The type accessors given below map an iterator type to its specific kind of iterator. Not all tag classes are related in hierarchies. For example, the Boost Graph library includes tag classes that describe whether or not a graph will allow multiple edges connecting two vertices (i.e., whether or not a graph can be a multigraph). The tag classes `allow_parallel_edges` and `disallow_parallel_edges` are unrelated.

Tag classes represent the names of concepts, and when organized into hierarchies, inheritance is analogous to refinement. We say that they represent the names of concepts since they do not explicitly support any kind of constraint or type association feature. In essence, we are equating the names of concepts to properties of types.

### 3.5 Tag Dispatch

The *tag dispatch* programming idiom provides a framework for the compile-time selection of algorithms based on arbitrary properties of types. This is used, for example, to implement a number of iterator-based algorithms. Consider, the distance operation shown in Figure 37.
template<typename Iter>
type iterator_traits<Iter>::difference_type
type size_t do_distance(Iter f, Iter l, iterator_tag) {
    type iterator_traits<Iter>::difference_type n = 0;
    for( ; f != l; ++f, ++n);
    return n;
}

template<typename Iter>
type iterator_traits<Iter>::difference_type
do_distance(Iter f, Iter l, random_access_iterator_tag) {
    return l - f;
}

template<typename Iter>
type size_t distance(Iter f, Iter l) {
    type iterator_category<Iter>::type Category;
    return do_distance(f, l, Category());
}

Figure 37. The distance operation uses tag dispatch to select an optimal implementation based on the kind of iterator.

Tag dispatch relies on standard overloading techniques to cause the compiler to choose the best dispatch. Here, the distance operation passes an object whose type is determined by the iterator_category type accessor. This will be one of the tag classes in the hierarchy in Figure 36. The compiler orders the instantiated overloads with respect to the inheritance hierarchy. Any iterator tag convertible to the iterator_tag will be dispatched to the first overload, unless it is convertible to the random_access_iterator_tag, in which case the call is dispatched the second. As such, tag dispatch is analogous to concept-based overloading—the ordering of overloads based on the refinement lattice.
3.6 SFINAE Traps

The C++0x programming language introduces two new features that can be used to augment the metaprogramming capabilities of the current language. These are `decltype` and the extension of SFINAE to trap substitution failures for any expression. We use these language features to systematically define an idiom for systematically trapping and reasoning about substitution failures. This work is developed as part of the Origin C++0x Concept Emulation library [Sutton, Maletic 2010].

The `decltype` facility is a type accessor that evaluates to the declared type of an expression, and it can be used anywhere a type is expected to be written (variable declarations, parameter declarations, return type specifications, cast operations, etc.). By using `decltype` to deduce the declared type of an expression, the compiler must resolve (lookup) any of the operations included in that expression. This implies that a substitution failure can occur when these lookups are made with SFINAE context (i.e., instantiating a function signature for overload resolution). The extension of SFINAE covers lookups for operations in expressions.

This systematic implementation of this idiom builds on a number of the other idioms and language features discussed in previous sections. There are components of the idiom: a SFINAE trap, a type accessor, and a type predicate. The SFINAE trap catches lookup failures for a fixed expression, the accessor returns the deduced expression type, or a symbolic `lookup_failure` type, and the predicate determines if the deduced type is valid. This idiom is designed to evaluate these expressions without triggering
compilation errors, allowing sophisticated metaprograms to be developed against the structural features of types. An example implementation is given in Figure 38.

```cpp
template<typename T>
auto member_swap(T& x, T& y) -> decltype(x.swap(y));

lookup_failure member_swap(...);

template<typename T>
struct member_swap_result {
  typedef decltype(
    member_swap(declval<T&>(), declval<T&>())
  ) type;
};

template<typename T>
struct has_member_swap {
  static constexpr bool value = !is_same<typename member_swap_result<T>::type, lookup_failure>::value;
};
```

**Figure 38.** The *member_swap* traits determine if a type *T* defines a member variable named *swap*.

The SFINAE trap is implemented by the two *member_swap* overloads. The first overload is a function template declaration (writing using a C++0x late-binding return type: `auto and ->`) whose result type is the deduced type of the expression `x.swap(y)`. If the type *T* does not define a member named *swap*, then this expression generates a lookup failure, which is propagated through the *decltype*, ultimately causing the overload to be rejected for consideration.

The second *member_swap* overload is a *variadic function* (not to be confused with a variadic template) similar to the Standard C Library’s `printf` function. It accepts any number of arguments of any type, and returns a tag class named `lookup_failure`. The variadic overload is always a viable overload candidate. We further note that variadic
functions are the “weakest” in terms of function overload resolution. If any candidate is a closer match than a variadic function, the other candidate is selected first.

The SFINAE trap works as follows. When the compiler resolves a call to the function, it considers the two overloads. Because the first overload is a template, it is instantiated. If the instantiation results in a substitution failure, the compiler selects the variadic function, whose result type is `lookup_failure`. If no substitution failure occurs, then the compiler selects the template overload because it is “more specialized” than the variadic version.

The type accessor, `member_swap_result`, returns the deduced type of the SFINAE trap, `member_swap` for a type `T`. This is done by applying `decltype` to a function call of `member_swap`. The `declval` function template generates function arguments of the given type (here `T&` for both arguments) in order to match the function signature of the `member_swap` trap. In this case, we can usually expect the result type to be `void` on success and `lookup_failure` on failure.

The type predicate, `has_member_swap`, determines if the type `T` defines a member function named `swap`. This is done by simply determining if the `members_swap` result type is `lookup_failure` or not. If the deduced type of the expression is `lookup_failure`, the type `T` does not define that operation.

This idiom—the deduction and testing of expression types—is essentially the underlying mechanism used to automatically check required interfaces with concepts. It is possible to map each associated function requirement in a concept definition to an instance of this idiom, and expressing a concept check as a conjunction of interface tests.
For example, the member swap facilities given here correspond to a concept whose name might be `Member_Swappable`.

### 3.7 Partial Template Definition

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 definition* 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 39.

```cpp
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 39.** 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 does not match a specialization, 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, as seen in Figure 17.
This idiom does not specifically correspond to any language features in C++0x, nor is it discussed in previous literature. Rather, it is a technique used to restrict or manipulate template instantiations and is the basis of the concept-controlled polymorphism idiom discussed in Section 3.8.

### 3.8 Concept-Controlled Polymorphism

As previously discussed, we can use SFINAE to affect how the compiler selects function overloads (shown in Figure 5). Concept-controlled polymorphism [Järvi, Willcock, Lumsdaine 2003] provides a systematic way of using SFINAE to control overloading based on arbitrary properties of types (i.e., concepts). This idiom is encapsulated in the `enable_if` class template, whose definition shown in Figure 40.

```cpp
template<bool, typename T = void>
struct enable_if_c {
    typedef T type;
};

template<typename T>
struct enable_if_c<false, T> {
};

template<typename Pred, typename T = void>
struct enable_if
    : enable_if_c<Pred::value, T>
{  
};
```

**Figure 40.** The `enable_if` template defines a result type if the boolean predicate is true. No nested type is defined otherwise.

The `enable_if_c` metafunction is a partial template definition that evaluates a type predicate. If the predicate evaluates to true, `enable_if_c` is defined as a type accessor, exposing a nested type declaration. If false, `enable_if_c` is an empty class. The `enable_if` template simply inherits the results of that specialization for a
metafunction or type predicate, \texttt{Pred}. When partial template’s result \texttt{type} is referenced in a substitution failure context (i.e., function header or class template specialization), it can trigger a substitution failure, eliminating the overload from the candidate set. Its counterpart, \texttt{disable_if} simply negates the logic of \texttt{enable_if}.

The enablement and disablement of algorithms must be done carefully in order to prevent the compiler from instantiating ambiguous overloads. Let us revisit the \texttt{swap} algorithm given in Figure 5 and correct the ambiguities discussed previously. An unambiguous implementation is given in Figure 41.

```cpp
template<typename T>
typename enable_if<has_member_swap<T>, void>::type
swap(T& x, T& y) {
    x.swap(y);
}

template<typename T>
typename disable_if<has_member_swap<T>, void>::type
swap(T& x, T& y) {
    T z = x;
    x = y;
    y = z;
}
```

\textbf{Figure 41.} The \texttt{sort} algorithm uses concept-controlled polymorphism to select either the canonical \texttt{swap} algorithm or a member-specialized variant.

Here, the \texttt{enable_if} and \texttt{disable_if} templates are applied to the return types of the two \texttt{swap} overloads in such a way that only one overload will ever be selected. If the type \texttt{T} supports a member swap operation, the first overload is enabled and the second is disabled. Conversely, if \texttt{T} does not support a member \texttt{swap} operation, then the second overload will be enabled and the first disabled.
This idiom parallels the application of the `requires` clause to constrained templates. This would be no different than requiring the `Member_Swappable` concepts or its negation.

### 3.9 Template Metaprogramming

In this section we give a brief overview of template metaprogramming techniques that are common in C++ generic libraries. A thorough exploration of the technique is given in [Abrahams, Gurtovoy 2005]. Template metaprogramming is predicated on the use of class templates to construct compile-time logic. The techniques used here are no different than the techniques used to define type traits; they are actually closely related. However, effective metaprogramming necessitates the definition of basic data types for the “language” in which metaprograms are written. To begin, we present the definition of the boolean metaprogramming type and its values. These are shown in Figure 42.

```cpp
template<bool Value>
struct bool_ {
    typedef bool_<Value> type;
    static constexpr bool value = Value;
};

typedef bool_<true> true_;
typedef bool_<false> false_;
```

**Figure 42.** The `bool_` class template is used to represent compile-time boolean values as constant metafunctions.

The `bool_` class template is used to project the boolean values `true` and `false` into the type system. This means that `bool_<true>` and `bool_<false>` can be used with the myriad of compiler features for operating on types—especially pattern matching to
support overloading and class template specialization. The type aliases `true_` and `false_` are often used as shorthand for `bool_<true>` and `bool_<false>`.

A metafunction looks like a combination of type accessor and type predicate, defining both a type and value component. However, the intent of these definitions is somewhat different. Metafunctions prefer to use the nested type to represent the compound “value” or state of the metafunction. The nested value declaration is provided to allow the compiler to evaluate integral and/or boolean constants. This technique can be used to build the standard logical operations, and_, or_, and not_. The logical not_ metafunction is trivial to implement. Its definition is given in Figure 43.

```cpp
template<typename Arg>
struct not_
  : bool_<!Arg::value>
{ }
```

**Figure 43.** The not_ metafunction defines a method of computing the logical not of boolean metafunction constants true_ and false_.

The use of inheritance to implement metafunction computation simply prevents metaprogram authors from writing multiple nested declarations. The template parameter, Arg, must also be a metafunction, minimally defining a nested value constant. Most importantly, this allows the metafunction to interoperate with boolean predicates defined in Section 3.2. The definitions of the variadic and_ metafunction is substantially more involved and is given in Figure 44.
template<typename... Args> struct and_;

template<typename Arg>
struct and_<Arg>
    : Arg
{ };

template<typename Arg, typename... Rest>
struct and_<Arg, Rest...>
    : bool_<Arg::value && and_<Rest...>::value>
{ };

Figure 44. The variadic and_ metafunction computes the logical conjunction of its arguments. Each argument is required to be a boolean metafunction.

Here, the and_ metafunction is defined as a variadic template, allowing programmers to compute the logical conjunction of a sequence of metafunctions. The result is computed by recursively applying the and_ metafunction to the arguments and then and-ing the results as an argument of a new boolean metafunction. Like the not_ metafunction, arguments are also required to be integral metafunctions (defining a nested value). Note that the and_ metafunction correctly short-circuits. If the left-most argument is false, the compiler is not required to evaluate (i.e., instantiate) the remaining arguments.

Template metaprogramming is practically ubiquitous (although not always obvious or extensive) in most generic libraries. The and_ and not_ metafunctions are specific examples that are used heavily in the Origin C++0x Concept Emulation Library described in Chapter 4.

3.10 Mixins

A mixin is a class that can be used to “inject” functionality into a user-defined type [Smaragdakis, Batory 2001; VanHilst, Notkin 1996]. 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.

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 45 shows an abbreviated version of the Boost archetype that enables tests for default and copy constructability.

```cpp
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 45. The default constructible archetype uses the mixins pattern to support interface aggregation.**

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 by nesting one archetype class as template arguments of the other, resulting in the type `copy_ctor_arch<def_ctor_arch<>>`. The resulting class should expose exactly both constructors, but no more.

We note that this idiom is largely a design technique and does not correspond to any proposed language features. However, it is still a useful method for building generative
data structures [Czarnecki, Eisenecker 2000; Smaragdakis, Batory 2001] or policy-based classes [Alexandrescu 2001].

3.11 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 [Coplien 1995]. 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 adaptive technique that is frequently used to simplify the process of adapting user-defined types to a known interface.

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.
template<typename Derived, typename Value>
class iterator_facade {
    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 46. 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.

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 most important feature of this class is the self function, which statically downcasts this object to its Derived class. Note that most instances of CRTP to not implement self. This is given for convenience. 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.
As with mixins, this programming is not specifically addressed by a new language feature. We do note that in some respects, CRTP is similar to Haskell’s type classes’ adaptive capabilities [Bernardy et al. 2008; Hall, Hammond, Jones, Wadler 1994]. Specifically, they define and provide a known interface, but require the user to provide at least some part of an implementation.
CHAPTER 4

EMULATING CONCEPTS

In this chapter, we present an overview of the Origin C++0x Concepts Emulation Library. This library was originally designed and implemented to provide features of concepts and constraints for experiments with generic library design. Ultimately, the work building these libraries serve as a validation of the mapping from programming idioms to language features described in Table 1. This validation demonstrates that concepts could be largely implemented as a set of source-to-source transformations, not unlike the operation of the original C++ compiler (CFront) [Stroustrup 1983]. This is also validated in [David, Haveraaen 2009], who describe concepts as syntactic sugar for an idiomatic framework that is not entirely dissimilar from the work presented in this chapter.

4.1 Emulation Techniques

We begin by making some observations about concept requirements in C++0x. Consider the concept identifier `Same_Type<T, U>`. Here, `Same_Type` is concept name, and the types `T` and `U` are its template arguments being validated by the concept’s predicate. Since concepts no longer part of the language, this expression simply becomes a template instantiation, and `Same_Type` is required to be a template with `T` and `U` its template arguments. In the Origin Concepts Emulation Library, we implement concepts as class templates, called a concept class. We say that the template specialization
Same_Type<T, U> is a model of the concept Same_Type. If T and U satisfy the requirements of the concept, then the model is valid. Otherwise, the model is invalid.

The implementation of each concept class requires the definition of two components: a nested type predicate named check that evaluates whether or not the model is valid and a nested (non-template) class assertion that, when instantiated, will result in a static assertion (static_assert is a new feature in C++0x) if the check fails. These two nested classes support the compile-time evaluation of type properties and the ability to generate concept-specific error messages.

4.1.1 Automatic Concepts

A concept class is automatic if it contains metaprogram logic to evaluate its own requirements. Consider the definition of the Same_Type automatic concept shown in Figure 47.

```cpp
template<typename T, typename U>
struct Same_Type {
    typedef is_same<T, U> check;

    struct assertion {
        ~assertion() { 
            static_assert(
                check::value, "Failed concept check SameType<T, U>");
        }
    };
};
```

Figure 47. The Same_Type concept is implemented as a class template with a nested check metafunction and assertion class.

The associated check defines (but does not evaluate) a type predicate. Because the predicate is not evaluated, the concept class itself can always be instantiated as model. This requirement of concept class definition ensures that simply writing an identifier to
describe a model does not accidentally result in compiler errors, although there are rare cases where compilation failures are unavoidable. In order to evaluate whether the model is valid, we instantiate the type predicate by writing, \texttt{Same\_Type<T, U>::check::value}.

The \texttt{assertion} class helps to provide meaningful error messages if the model is not valid. Instantiating the destructor of this class simply asserts the \texttt{check} predicate. Because this is a destructor, its instantiation necessitates the instantiation of any base class destructors of the assertion class. This is the case when the concept has refinements. Examples are given later in this chapter.

### 4.1.2 Explicit Concepts

Explicit concepts can only result in valid models if a user defines a concept map for the specified type or set of types. Without a model or concept map declaration, any evaluation of the concept will result in an invalid model. Consider the \texttt{Forward\_Iterator} concept class defined in Figure 48.

```cpp
template<typename X>
struct Forward\_Iterator : Iterator<X> {
    typedef false\_ check;
    struct assertion {
        ~assertion() {
            static assert( check\::value,
                “No Forward\_Iterator\<X\> concept map declared”)
        }
    };
};
```

Figure 48. A definition of the explicit concept \texttt{Forward\_Iterator} is always invalid, unless explicitly declared to be otherwise.
Here, check function is simply defined as the constant metafunction false_ (whose nested value is always false), and the assertion will result in an error message that indicates both the failure and the reason for that failure.

In the Origin Concepts Emulation Library, the definition of concept maps is accomplished using (partial) class template specialization of a concept class and redefining the concept check and assertion to indicate that the model is valid. The concept map specialization of Forward_Iterator concept for pointer types is shown in Figure 49.

```cpp
template<typename T>
struct Forward_Iterator<T*>
 : True<true>
{ }
```

Figure 49. The Forward_Iterator<T*> specialization denotes the satisfaction of the Forward_Iterator concept by all pointer types.

Here, the concept map is derived from the model True<true> (True is a concept), which provides the necessary and trivially true check and assertion functions. When the concept is checked, the specialization is looked up instead of the concept class. Since the concept map refines a valid model, it will always result in an affirmative check. We say that the concept map affirms the model. We also note that a concept can be concept can be explicitly negated by inheriting from True<false>, a contradiction. In the case that the concept requires the definition of associated types, these can simply be written in to the concept map, much like a traits class specialization.

We describe the use of explicit concepts and affirmations (or negations) as adaptive modeling. More formally, adaptive modeling is the technique of using concepts and
concept maps to adapt types to those concepts. This is the technique use to adapt pointers to the `Iterator` concept hierarchy, and this would also be the technique used to develop and adapt graph types for the Boost Graph Library. We observe two basic reasons for designing concepts using this technique. First, explicit concepts are required when a concept is predicated on a *semantic* requirement or axiom. This is essentially unavoidable. The second reason is by choice. Our experience in developing these techniques and this library has shown that we can largely eliminate the need for explicit adaptive models by effectively constructing the concept checking features of concept classes. Since associated type requirements are largely deduced from the result types of operations, adaptors need only provide the requisite overloads in order for the concept check to succeed. Based on these results we do not believe that extensive use of explicit concepts to achieve adaptability is a good design choice for generic libraries.

The use of template specialization to create concept maps can create “gaps” in the concept checking features of the library. This is a known limitation of the approach. Because the checked requirements are defined in the concept class, concept map specializations can simply bypass those checks. In some cases, especially when creating adaptive models, requirements should not be bypassed. Extracting the deductive components of the `check` predicate from the concept class and then re-applying it within the concept map could solve this problem, but the technique has not been thoroughly evaluated.
We note that no technique was defined to emulate the explicit refinement feature of
concepts. However, we do speculate that this can be trivially done by simply excluding
the checks and assertions in the refined concept class.

4.1.3 Requiring Operations

Unfortunately, there is no simple analog for the style of expressing required
operations that was proposed for C++0x concepts. However, we can write type traits that
determine the validity of arbitrary expressions as described in Section 3.6 (SFINAE Traps). Consider the Equality_Comparable concept class shown in Figure 50.

```
template<typename T>
struct Equality_Comparable {
    typedef and_<
        has_equal<T, T>, has_not_equal<T, T>
    > check;
};
```

**Figure 50.** The Equality_Comparable concept requires `T` to support comparisons via `operator==` and `operator!=`.

Here, the check metafunction evaluates two conditions, namely that `T` defines`operator==` and `operator!=`. These predicates refer to trait implementations similar
to the has_member_swap in Figure 38. In some cases, a concept may define large
number of operational requirements to evaluate, sometimes including guarantees about
the result type of those operations. In such cases, it is sometimes convenient to write the
checks as template metafunctions outside the scope of the concept class.

4.1.4 Defining Traits

The typename declarations within a concept definition are directly analogous to the
associated types of a traits class. The concept decouples access of the associated type
from its concept argument(s). In many cases, the actual types of these declarations are deduced as the result type of an operator. Concept classes in the Origin Concepts Emulation Library are constructed similarly, but rely on type traits to implement the deduction. Consider the Has_Plus concept shown in Figure 51.

```cpp
template<typename T, typename U>
struct Has_Plus {
  typedef plus_result<T, U>::type result_type;
  typedef has_plus<T, U> check;
};
```

**Figure 51. The Has_Plus concept contains a deduced type result for the addition of types T and U.**

A typename declaration is realized as a typedef by the concept class. The plus_result metafunction will return the result of the operation if it exists. If the operation does not exist, the result_type will be lookup_failure due to the SFINAE trap discussed in Section 3.6, and the check metafunction will result in false. The absence of an operator+ for types T and U will not result in a compilation error.

Deriving type names from concept arguments is also a common practice (e.g., an iterator’s reference type). In these cases the typedef could simply alias the associated type or use an accessor metafunction if more variability in the underlying type structure is allowed.

### 4.1.5 Refining Concepts

Concept refinement is implemented using inheritance. Refining from other concepts requires the usage of those refinements in the check metafunction and assertion class. The Has_Virtual_Destructor concept class is shown in Figure 52. Here, the
refining concept class inherits the refined concepts and their requirements. Refinement imposes two additional requirements on the concept provider: explicitly checking refinements in `check` metafunction, and inheriting the refined `assertions`.

```cpp
template<typename T>
struct Has_VirtualDestructor
 : HasDestructor<T>, Polymorphic_Class<T>
{
  typedef and_<
    has_virtualDestructor<T>,
    concept_check<
      HasDestructor<T>, Polymorphic_Class<T>
    >
  > check;
  struct assertion
    : HasDestructor<T>::assertion,
      Polymorphic_Class<T>::assertion
    { ... };
};
```

**Figure 52.** The `Has_VirtualDestructor` concept class refines `HasDestructor` and `Polymorphic_Class` and provides specialized concept-checking logic in its `check` and `assertion` components.

Note that the `check` metafunction, shown in Figure 52, calls a metafunction, `concept_check`, which simply evaluates a conjunction of the given concept class arguments’ check predicates.

### 4.1.6 Nesting Requirements

Associated requirements are implemented similarly to refined requirements. Nested requirements typically establish constraints on associated types or build relationships between those types and the concept arguments. Consider the `Semiregular` concept, shown in Figure 53.
template<typename T>
struct Semiregular
 : Copy_Constructible<T>, Copy_Assignable<T>
{
    typedef Same_Type<
        typename Copy_Assignable<T>::result_type, T&
    > Req;

    typedef concept_check<
        Req, Copy_Constructible<T>, Copy_Assignable<T>
    > check;

    struct assertion
     : Copy_Constructible<T>::assertion,
      Copy_Assignable<T>::assertion
    {
        assert_requirements<Req>();
    };
};

Figure 53. The requirements for the Semiregular concept are implemented as a nested, named requirement that is checked similarly to its refinements.

The Semiregular concept’s nested requirements are implemented simply by creating a typedef for the requirement, and checking them with concept_check. In the concept class’ assertion, the assert_requirements function causes the recursive instantiation of the destructors of the given requirements.

4.1.7 Constraining Templates

Since C++ lacks the syntax for imposing type constraints on templates, writers of providers of generic algorithms and data structures must work within the confines of the existing language mechanics to provide these features. Essentially, we have to rely on concept-controlled polymorphism, which is encapsulated by the enable_if facility [Järvi, Willcock, Lumsdaine 2003]. The Origin Concepts Emulation Library supports two different facilities for constraining templates: the concept_assert and
concept_enable metafunctions. The concept_assert metafunction statically asserts that a model is valid, resulting in a compiler error if it is not. The concept_enable metafunction is a type predicate that works like enable_if, but takes models as arguments instead of arbitrary type predicates or metafunctions. In the Origin Concepts library, the technique for constraining function templates primarily relies on the definition of additional template parameters. Consider the signature of the find algorithm shown in Figure 54.

```cpp
template<
    typename Iter, typename T,
    typename = typename concept_assert<
        Iterator<Iter>,
        Has_Equal<typename Iterator<Iter>::value_type, T>
    >::type
> Iter find(Iter f, Iter l, T const& x) {
    for( ; f != l; ++f)
        if(*f == x) return f;
    return l;
}
```

**Figure 54.** A constrained implementation of the `find` algorithm implements its constraints as a default template parameter.

Here, the constraint is implemented as an unnamed, default template parameter for the algorithm. By writing the constraint as a template parameter, it does not obscure the writing of the function signature in the same way that writing enable_if as a return specifier or function parameter might. The concept_assert metafunction applies constraints to its arguments, the models `Iterator` and `Has_Equal`. If either of the requirements is not met, then a static assertion will cause compilation failure, emitting a specific error message. Concept assertions can be applied to class templates and member function templates in exactly the same way.
Note that using `concept_assert` will result in a compilation error. Since there are no overloads of the find algorithm that share this signature (i.e., having the same parameter and return types), the use of an assertion is preferable to a simple check (and possible SFINAE elimination). If the algorithm were constrained using `concept_enable` and any of the requisite models failed the check, then the function would be quietly removed from the overload candidate set, and the compiler would simply report an error that no overload could be found. In short, `concept_assert` should be used when no overloads are present, and `concept_enable` (and `concept_disable`) should be used to select between similar overloads. In the case of multiple ambiguous overloads, the `concept_enable` and `concept_disable` features must be applied in the function signature (just like `enable_if`) since the definition of alternative defaults in template parameter list is insufficient to differentiate overloads.

The same techniques can also be applied to solve the concept-overloading problem. However, the technique used by the Origin Concepts library does not involve the partial ordering of overloads to select functions based on the most-refined concept. Consider the concept-overloaded `advance` declarations shown in Figure 24.
template<typename Iter>
void advance(Iter& i, int n = 1,
    typename concept_enable<
        Iterator<Iter>, concept_not<Bidirectional_Iterator<Iter>>
        >::type* = nullptr);

template<typename Iter>
void advance(Iter& i, int n = 1,
    typename concept_enable<
        Bidirectional_Iterator<Iter>,
        concept_not<Random_Access_Iterator<Iter>>
        >::type* = nullptr);

template<typename Iter>
void advance(Iter& i, int n = 1,
    typename concept_enable<
        RandAccIterator<Iter>
        >::type* = nullptr);

Figure 55. The three overloads of the iterator advance function. Overload differentiation is done using filtering expressions on the iterator hierarchy as opposed to tag dispatch.

recall that a C++0x requires clause is essentially a boolean expression on the validity of models. Without a modification to the partial ordering, a random access iterator will satisfy the requirements of both overloads and should result in ambiguous overloads. However, we note that simply rephrasing the query allows us to write constraints that exactly capture the semantics of the overload. Specifically, we choose to interpret the query: “If Iter is an Iterator but not a Bidirectional_Iterator then select the first overload. If Iter is a Bidirectional_Iterator, but not a Random_Access_Iterator, then select the second overload. Otherwise, if Iter is a Random_Access_Iterator, then the third overload must be selected. In essence, we can construct filtering expressions for the kinds of types used for each overload.

Conventionally, tag dispatch (as discussed in Section 3.4) would be used to implement this feature. However, this technique requires us to map every type to a
“most-derived” concept; in some cases this would require type providers or library users to define specializations or overloads of the mapping metafunction or function (e.g., `iterator_category` in Figure 36). This approach enables the concept-overloading to be deductive rather than proscriptive. It imposes little to no effort on the part of end programmer.

4.2 Implementation

The Concepts Emulation Library is currently a core component of the Origin C++0x Libraries. The Origin C++0x Libraries are a collection of programming utilities, data structures, and algorithms (not dissimilar from the C++ Standard Library) that are written in C++0x and used to support experiments with generic programming and library design. The architecture of the Origin C++0x Libraries is shown in Figure 56. The Origin.Core libraries (Meta, Traits, Utility, and Concepts) provide the foundation for many the abstract components (Iterators, Containers, Functions, and Algorithms). Although these are pictured as a layered architecture, they are mostly interdependent.

![Figure 56. The Origin C++0x Libraries are modeled on the C++ Standard Library and includes components for template metaprogramming, type traits, and concepts at its core.](image-url)
These libraries are implemented using the subset of features supported by the development branch of GCC (v. 4.5 as of this writing). Critical language features supporting these libraries include variadic templates, decltype, auto, lambda expressions, and the extension of SFINAE to arbitrary expressions. The Origin C++0x libraries do not use the C preprocessor outside of header guards. The intent is to provide a reference implementation for the techniques used for generic programming and library design, even in cases where the preprocessor could be used to simplify the reading and writing of source code.

The Meta and Traits libraries contain sufficient features to implement concepts, although they are not as feature complete as, e.g., Boost.MPL. The Traits library builds on the C++0x Standard Library type traits by providing capability and result type queries for all overloadable operators and special functions. The Concepts library implements most of the concepts described in the last C++0x Working Draft to concepts [Becker 2009], with some small variations. The Iterators library defines iterator concepts, and their operations (distance, advance, next, and prev). The Functions library defines a number of classes used to create adaptable function objects (functors) such as unary_function and binary_function and a number of useful functors (e.g., identity_of, less, etc.). The Algorithms library implements a subset of algorithms described for the C++0x Standard Library and constrained by the concepts in the Core libraries. The Containers library contains only concept definitions, but no container implementations. We are in the process of analyzing the structure of these containers in order to better evaluate the current design of the Container concept hierarchy.
Each library is validated by a test suite that assesses the correctness of its components. Test cases exercise and assert components to ensure, for example, that type traits and concepts are valid or invalid when expected.

4.3 Discussion

The exercise of developing this library provides a great deal of insight into the development of language features for concepts and several aspects of conceptual design techniques for C++ generic libraries.

4.3.1 The Nature of Concepts

One of the most successful aspects of this implementation is that concepts are treated as a kind of “advanced type trait”. In essence, a concept class encapsulates the logic that is used to determine whether or not a type conforms to the concept’s specifications. This take on concepts is very similar to the approach used in [Stepanov, McJones 2009]. In this work, the authors define concepts explicitly as the conjunction of mathematically expressed constraints and axioms. They do not use the concept syntax proposed for C++0x. In fact, we view the C++0x concept syntax as a grammatical façade to Stepanov’s underlying mathematical notation.

It is also worth noting that Stepanov’s treatment of concepts does not include refinement, only aggregation. Interestingly, the use of inheritance in the Origin Concept Emulation Library is purely for notational convenience. Refinement can be effectively implemented as requirement aggregation with little no effect on client usage.
We also note that the use of type traits to check interface requirements aligns well with the Texas A&M proposals for concepts since the SFINAE traps used to implement the checks actually instantiate (or fail) valid expressions. Furthermore, this approach favors automatic checking over explicit checking. This approach runs counter to many of the early proposals or ideas for concepts where explicit checking was assumed to be the norm [Gregor et al. 2006; Siek 2005; Siek et al. 2005; Siek, Lumsdaine 2005a; Zalewski, Schupp 2009]. Only late in the evolution of the concepts proposal was the idea of default automatic checking discussed [Stroustrup 2009b].

In this implementation, less emphasis is placed on the use of concepts as a generalized adaptation mechanism. Experience gained during the implementation of this library indicates that attempting to use concepts as an adaptive framework may not a sound approach for software design. To do so simply shifts the burden of building adaptors from one language features (classes) to another (concepts). The design space for building adaptors using classes is very well explored [Gamma, Helm, Johnson, Vlissides 1995], while the latter is less well understood [Järvi, Marcus, Smith 2007].

4.3.2 Casual Modeling

We say that a type casually models a concept if it “unintentionally” satisfies the concept’s syntactic but not semantic or intentional requirements. Here, we refer to intentional requirements as the domain-specific intent of the concept as meant by the concept provider, which cannot be easily expressed in a syntactic manner. The problems of casual modeling arise from implicitly or automatically checked concepts.
Casual modeling is especially problematic when two concepts in the same refinement hierarchy are differentiated only by semantic requirements (i.e., axioms). For example, the `Forward_Iterator` concept is required to be explicit because it is differentiated from `Iterator` by the `Multipass` axiom. Since it is currently infeasible to determine at compile time whether or not an axiom is satisfied, it is not possible to differentiate the two concepts. The `Forward_Iterator` concept must be declared explicit. In fact, this is the only aspect of the `Forward_Iterator` concept that must be explicitly declared in order for types to model the concept. This has substantial impact on the design of the concept hierarchy.

We favor an approach where explicit concepts are minimally specified. This means that concept providers should encapsulate aspects of concepts that cannot be deduced as concepts themselves. We may prefer to refactor the `Forward_Iterator` concept into two concepts: `Forward_Iterator` and `Multipass_Iterator`, shown in Figure 57.

```cpp
template<typename X>
struct Multipass_Iterator {
    typedef false_check;
};

template<typename X>
struct Forward_Iterator : Iterator<X> {
    typedef and_<
        concept_check<
            Multipass_Iterator<X>, Input_Iterator<X>
        >
    > check;
};
```

Figure 57. The `Forward_Iterator` concept is decomposed into two concepts: `Multipass_Iterator` and `Forward_Iterator`. 
The benefit of this approach is that \texttt{Forward\_Iterator} is made implicit, meaning that its refinements and other requirements must be explicitly checked, and \texttt{Multipass\_Iterator} concept encapsulates only a single, un-checkable property (axiom) of the types. We refer to the \texttt{Multipass\_Iterator} as an \textit{axiomatic concept}. Note that this approach to the design of concept webs maintains the “viral” property of explicit concepts. Since \texttt{Multipass\_Iterator} is an explicit concept, every type that models \texttt{Forward\_Iterator} or a refinement thereof must explicitly declare a concept map for this concept. We have experimented with this technique using a branch of the Origin Concept Emulation library and found that it is an effective and graceful solution to the problem of explicit concepts.

One downside to this technique is that some concepts may require a number of axiomatic concepts (e.g., \texttt{Less\_Than\_Comparable}). This would require type providers to declare a number of concept maps for each axiom required by the concept. In this case, the axiomatic concepts could be combined into a single concept via refinement or aggregation, or the concept provider could simply make the aggregating (automatic) concept explicit.

\textbf{4.4 Conclusions}

In this chapter, we have presented a number of techniques for emulating concepts using the language features present in the C++0x programming language and shown the viability of the technique by implementing. The Origin Concept Emulation Library provides a foundation for the creation of generic data structures and algorithms in the Origin C++0x libraries. The experience gained from this exercise yields insight into the
nature of the concepts language semantics and techniques for the design and use of concepts as constraints. Having established a basis for programming with concepts, we turn our attention to understanding the composition of generic libraries, and how these idioms are used in their design and implementation.
CHAPTER 5

REVERSE ENGINEERING GENERIC SOURCE CODE

This chapter presents the techniques and tools developed to reverse engineer, analyze, and study C++ generic libraries. Having described the techniques used in the construction of generic libraries, we now seek to understand how those techniques are used to build software, and ultimately how we can build tools to directly support programmers in maintenance tasks. This chapter presents several of the technical contributions of this dissertation, especially in the area of techniques for reverse engineering and source code modeling.

5.1 Related Work

Reverse engineering [Chikofsky, Cross 1990] and source code analysis is the cornerstone of a wide variety of applications supporting software maintenance especially in the area of program comprehension. Examples of engineering tasks supported by these applications include design recovery [Biggerstaff 1989], concept assignment [Biggerstaff, Mitbander, Webster 1994; Rajlich, Wilde 2002], feature location [Chen, Rajlich 2000; Eisenbarth, Koschke, Simon 2003]. Such tasks directly support a programmer’s comprehension of source code. Of particular relavence to this work are applications of design recovery, which frequently focus on the reverse engineering of source code into UML [Sutton 2005; Sutton, Maletic 2007b; Tonella, Potrich 2001; 2002; 2003].
A wide range of source code models supports these user-centric reverse engineering-based applications. For example, call graphs are a common abstraction and used for a number of applications including impact analysis [Murphy, Notkin, Griswold, Lan 1998]. Another technique, program slicing [Weiser 1981] is predicated on the system dependence graph, which supports some applications of feature location [Chen, Rajlich 2000]. The C Preprocessor has also been modeled to reason about control flow and source code configuration [Favre 1996; Snelting 1996; Sutton, Maletic 2007a]. Class inheritance hierarchies are also a common artifact in design recovery [Sutton, Maletic 2007b; Tonella, Potrich 2001] and the foundation of several analyses [Snelting 2000; Tip, Cho, Field, Ramalingam 1996] supporting comprehension and software maintenance. These source code models are often integrated into frameworks for reverse engineering [Ferenc et al. 2001; 2002; Müller, Klashinksy 1988; Müller, Orgun, Tilley, Uhl 1993; Premkumar, Brachman, Selfridge, Ballard 1990; Wong, Tilley, Müller, Storey 1995], visualization, and program transformation [Baxter 1992; Baxter, Pidgeon, Mehlich 2004] to support a wide range of programming and maintenance tasks.

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 [Gregor, Schupp 2005] 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 [Zalewski, Schupp 2006b]. An approach to support the debugging of template metaprograms is given in [Porkoláb, Mihalicza, Sipos 2006].
Although these applications incorporate some aspects of reverse engineering tools to support maintenance, they target on lower-level template or concept usage. They do not directly support large-scale program comprehension for C++ generic libraries.

5.2 Reverse Engineering Templates and Concepts

In this section, we describe a program model (or abstract syntax graph, ASG) for templates and concepts and the implementation of a reverse engineering framework built on this model.

5.2.1 An Program Model for Templates and Concepts

The model presented here is the foundation of the abstract source code models described in Section 5.3. At the core of this model is the representation of templates and their relationships. Figure 58 shows a UML description of the part of program model defining templates and template specialization.

![Figure 58. The C++0x program model that supports template elements and template specialization.](image-url)
In this model, a template is a scope that can be attached to other scoped elements (i.e., classes, functions, or concept maps) in order to create a template definition (as indicated by the template association between the Scope and Template elements). A template scope defines a sequence of zero or more template parameters. The definition of explicit or partial specializations is managed through a relation element named Specialization that associates each a specialized template definition with its primary template over a sequence of expressions, denoting a specialization pattern. The Scope, if a Template, will contain a set of specialization objects. In this model an Instance denotes a template specialization (the instantiation of a concrete type) over a sequence of template arguments, and relating to the template definition. The program model also includes an abstract specification of concepts, which is depicted in Figure 59.

**Figure 59.** The subset of the C++0x program model that supports concepts, concept maps, and refinement.
In this model, a *Concept* is a scoped element containing a sequence of *TemplateParameter* elements. A concept also defines a sequence of *Refinements* and *Constraints* (associated requirements). A *Refinement* is a relation between a general and refined concepts (much like inheritance). The refinement is defined by a *Requirement*, which is structurally analogous to a template *Instance*, but indicates an evaluable concept check. The requirement can also be negated. A Constraint is a conjunction of (possibly negated) *Requirements*. A *ConceptMap* is also a *Scope* defined over a specialization pattern. Each *ConceptMap* refers to its defining Concept as its model. Finally, all scoped elements (i.e., functions, member functions, classes, and even concept maps) can be constrained by attaching a Constraint element to them.

A hierarchy of template parameters is shown in Figure 60. A *TemplateParameter* is variadic, if its pack attribute is true, and they can be optionally constrained (i.e., representing the use of shorthand constraint notation). A *TypeParameter* represents template parameters introduced by a typename declaration. A *NonTypeParameter* is introduced by a variable declaration (e.g., `int N`). A *TemplateTemplateParameter* is introduced by a class template declaration.
The names of templates, template parameters, and concept maps are given special consideration in our analysis. A careful approach to the naming of templates and template parameters provides a useful framework for succinctly describing the parameterization of templates and template specializations. In C++, the names of template parameters (much like function parameters) are not required to be consistent between a declaration and definition. Consider the example in Figure 61.

```cpp
template <typename, int> class Array;

template <typename ValueType, int Size>
class Array {
    array();
    template <typename Iter> array(Iter, Iter);
};

template <typename T, int N>
array<T,N>::array() {
}

template <typename T, int N>
template <typename Iter>
array<T, N>::array(Iter f, Iter l) {
}
```

Figure 61. The names of template parameters are allowed to vary between declarations and definitions or omit them entirely.
Because of the allowable naming ambiguities, our analysis assigns symbolic names to each template parameter according to its kind and the depth of its template scope. For example, the first template parameter (ValueType or T) will be assigned the identifier $T_{1-1}$. The second parameter (Size or N) is named $N_{1-2}$. We use the ‘$’ as part of the template parameter name to help avoid collisions with other non-template parameter identifiers. The ‘T’ is used to denote type parameters, whereas ‘N’ is used for non-type (or integral) parameter, and ‘X’ is used for template template parameters. The numbering is used to indicate the canonical position of the template parameter within a nested set of template scopes. Here, 1-1 and 1-2 indicate that the nesting is at the first depth and the 1\textsuperscript{st} and 2\textsuperscript{nd} position, respectively. The Iter parameter in the template array constructor is named $T_{2-1}$ because it is the first parameter in the second tier of template scopes. A template parameter can be declared as a parameter pack to create a variadic template, by using the ellipsis (…) specifier. A parameter pack can be applied to any kind of template parameter. The name of a type parameter pack (e.g., typename... Args) might be given as $T_{1-1+}$. Template parameter packs do not currently receive any special consideration in our analyses; they are simply a different kind of template parameter, which can be used to help differential specializations.

The names of templates and template specializations are written in argumented form, which encodes the sequence of specialization matching pattern into the name of the class, even for base templates. The argumented form defines a consistent and uniform method for naming and distinguishing templates. Recall, from Section 2.1.2, the definition and specialization of the Vector data types (Figure 7, Figure 9, and Figure 11). Using the
augmented form, the name of the primary template is written as `Vector<$T1-1>`. The explicit specialization for bool is named `Vector<bool>`. The partial specialization over the `Compact_Value` class template (defined in Figure 10) is given the name `Vector<Compact_Value<$T1-1, $N1-1>>`. This style of naming can be used to determine if the name of a class template refers to a base template, an explicitly specialized template, or a partially specialized template. We further define the template-id of a template as the name of the template element, without additional arguments (e.g., `Vector`). The names of concept definitions are not written in argumented form (since they cannot be specialized), but concept maps are.

### 5.2.2 Implementation

The srcTools reverse engineering framework uses this model internally to reconstruct information about parsed source code. srcTools is a Python-based reverse engineering and fact extraction framework built on top of the srcML markup for C++. [Collard, Kagdi, Maletic 2003; Collard, Maletic, Marcus 2002] 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 a reverse engineering parser that aims to support lightweight tools and rapid application development, it forgoes many of the responsibilities traditionally associated with compilers. 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 [Cox, Clarke 2000; Sim, Holt, Easterbrook 2002]. As a result, the srcML translator is not able to disambiguate some aspects of the language and can generate inaccurate srcML output.

The srcTools framework is comprised of parsing and modeling components. The parser is capable of reconstructing an ASG from the XML input generated by the srcML translation, and implements several advanced features such as C++ name lookup and overload resolution. 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 ASG-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 ASG 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 [Gamma, Helm, Johnson, Vlissides 1995]. This is to say that Python modules interact with the parser framework by registering handlers for specific parsing or ASG-construction events. The internal construction of the srcTools ASG 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 `std::vector` class template would be found in the `src_class` table with the identifier `std::vector<$T1-1, $T1-2>`, where $T$ indicates a type parameter, and the $m-n$ notation describing its canonical position if the parameter rather than name. The naming scheme is described in detail in Section 5.2.1. Here, the second type parameter is an allocator. This particular naming of class and function templates enables srcTools to effectively differentiate templates and their specializations.

### 5.3 Source Code Models for Generic Libraries

A source code model is defined as a labeled (and often directed) graph over the elements of a program. For example, a call graph is a directed graph that connects functions along a call-relation. The control flow graph is a directed graph that connects statements or blocks according to the flow, loops and branches of a program. An inheritance hierarchy is a directed graph that relates derived classes to their bases.

One of the most challenging analytical problems caused by the pervasive use of templates (and concepts) is the way in which they decouple concrete program elements (i.e., types) from their abstract specifications (type parameters). Since type information is only propagated through templates during instantiation, the accurate recovery and description of program design from the source code is made very difficult. To better facilitate reverse engineering of such programs, we define several source code models derived from the program model defined in Section 5.2. Specifically, we define the template instantiation graph, the constraint propagation graph, and the concept lattice.
Parts of this work were presented in [Sutton, Holeman, Maletic 2009]. The primary contribution of this work to the dissertation is the validation of the reverse engineering model and techniques presented in Section 5.2. Moreover, the results of this work provide a basis for the development of new techniques for software maintenance or program comprehension tools.

5.3.1 Template Instantiation Graph

The *template instantiation graph* relates scoped elements to the templates that they instantiate. Formally, we define the instantiation graph as a directed multigraph $IG = (V,E)$. Each vertex is labeled with a class template name. A vertex $u$ is connected to a vertex $v$ if $u$ refers to a type that is an instantiation of the template labeled by $v$. The edge $(u, v)$ is labeled with the sequence of arguments supplied to the template instantiation.

If a template is found to have any specializations, then a *specialization vertex* is created and labeled with the template name, but not the arguments. Instantiations of specialized template are connected to the specialization vertex rather than the concrete template. For example, consider the code shown in Figure 62.
template <typename P> struct extract_policies {
    typedef typename P::type type;
};
template <typename... P> struct wrap_policies {
    typedef policies<P...> type;
};
template <typename... P> struct wrap_policies<
policies<P...>> {
    typedef policies<P...> type;
};
template <typename P> struct normalize_policy {
    typedef typename mp::eval_if<
        typename is_derived_policy<P>::type,
        extract_policies<P>,
        wrap_policies<P>::type
    >::type type
};
template <typename T, typename P>
class vector<T, P> : public vector_base<
    T, typename normalize_policy<P>::type
> { 
};

Figure 62. The extract, wrap, and normalize metafunctions are used to
determine the instantiation of vector_base for the given vector specialization.

In this program, the vector class template specialization normalizes the template
parameter P to ensure that the class is wrapped in a policies template. The resulting
instantiation graph is given in Figure 63. Here, the instantiations of the arguments to
eval_if metafunction are attached to the normalize_policy vertex. The
wrap_policy vertex represents optional instantiation of either specialization (not unlike
virtual functions in polymorphic classes).
Instantiations of nested class templates require special consideration. Consider an abbreviated version of the `rebind_allocator` metafunction shown in Figure 64.

```cpp
template <typename T, typename P>
struct rebind_allocator {
    typedef typename extract_allocator<P>::type A;
    typedef A::template rebind<T>::other_type;
};
```

**Figure 64.** The `rebind_allocator` metafunction uses the `extract` metafunction to retrieve an allocator type, and then rebinds it to the type T.

The metafunction uses a `typedef` to create a type variable. In our analysis we treat type variables as first-class generic program elements that can behave as classes or as class templates. The instantiation graph corresponding to this metafunction is given in Figure 65.
Figure 65. The instantiation graph of the `rebind_allocator` metafunction instantiates both `extract_allocator` and the `rebind` template of the type variable `A`.

We have implemented the reverse engineering and modeling of templates and their instances by extended the reverse engineering capabilities of the srcTools framework and srcfacts fact extractor to support the generation of these graphs. The resulting data is output to the srcTools relational database, from which the visualization is trivially mapped into the Graphviz toolset [Ellson et al. 2002; Gansner, North 2000] for rendering.

Beyond its obvious applications as a visual aid to program comprehension, the template instantiation graph will be invaluable to analyses that support metafunction debugging [Porkoláb, Mihalicza, Sipos 2006]. For example, a variant of this graph could be defined to model control flow for template metaprograms by attaching special semantics to selective (e.g., `eval_if`) or recursive templates. This graph could also be used to emulate data flow for template parameters. Control flow and data flow analysis are already widely used in a variety of software maintenance applications including impact analysis, program slicing, and feature location. Modeling the abstract structure of template metafunctions will provide similar facilities for reasoning about generic source code.
5.3.2 Constraint Propagation Graph

The concept propagation graph relates templates to concept definitions that are referenced via requires statements or refinements, and describes the propagation of template parameters through a constraint system. The construction of this model is nearly identical to that of the template instantiation graph, except that it is constructed over the requirements of concept definitions on template parameters. Vertices in the graph are constrained program elements (i.e., templates), and concept definitions. A directed edge connects an element \( u \) to an element \( v \) if \( u \) contains a requirement for the concept definition \( v \). The edge is labeled with the arguments of the requirements.

The constraint propagation graph for the InputIterator concept (a version of which is defined in Figure 20) is shown in Figure 66, although the constraints represented by the graph Figure 66 represent a slightly different implementation than the version given in Figure 20. Recall that InputIterator enumerates a number of requirements for both its template parameter \( x \), and the associated type \( s \) of \( x \). The constraint propagation graph also includes the refinements of the concept, since they are also requirements.
Figure 66. The constraint propagation graph for the Input_Iterator concept shows the propagation of type requirements across concept constraints.

One obvious application of the constraint propagation graph is impact analysis. By considering the reverse of this graph, one can determine the set of elements that will be impacted by a change in the requirements of a concept. This is critical application in the maintenance of generic libraries since changes to a concept definition can fundamentally alter the semantics of the abstraction.

5.3.3 Concept Lattice

The concept lattice is a source code model constructed over the refinement relation between concept definitions. Specifically, this graph is a subgraph of the constraint propagation graph that is only concerned with refinement relations (i.e., type propagation is not considered). The concept lattice for generic programming is directly analogous to the inheritance hierarchy for object-oriented programming.
One application of the concept lattice is its use as a visual aid to program comprehension (akin to an inheritance hierarchy). To demonstrate its applicability, we have modeled the C++0x support concepts, which describe the kinds of types in the C++ type system. This model, rendered using Graphviz is shown in Figure 67. This lattice provides a substantial amount of information about the C++ type system and the relationships between different kinds of types (e.g., the differentiation between `Class` kinds of types and `ValueType` kinds of types).

![Concept Lattice for C++ Type System](image)

Figure 67. The concept lattice for the C++ type system defines the basic kinds of types and type constructors available in the C++0x programming language.

The concept lattice model is also an extremely valuable artifact for automated analyses of generic libraries. The concept lattice is as a component of static analysis to support the impact analysis of the changes to a concept definition [Zalewski, Schupp 2005; 2006a]. Note that this work operates on unconstrained templates so the constraint
propagation graph cannot be used to support this application. Concept lattices are also used as an aid to support library analysis and transformations [Schupp, Gregor, Musser 2001; Schupp, Gregor, Musser, Liu 2002]. The Simplicissimus software uses concept hierarchies to support the semantic checking of operations and ensures the transformations remain conceptually consistent. The concept lattice is used in several different components of the analyses for constraint identification [Sutton, Maletic 2008]. In this work, the concept lattice is used to collect refined concept definitions and to determine the least common descendent (the meet) of a set of concepts. The application of the concept lattice to constraint identification is discussed in Section 8.2.4.

5.4 Conclusions

This chapter presents an overview of the program model used to represent templates and concepts by the srcTools reverse engineering framework, whose implementation is also described. Three source code models of are presented as an application of these reverse engineering and fact extraction technologies. These tools are used to conduct the empirical studies and source code analyses in the remaining chapters of the dissertation.
CHAPTER 6
IDIOM USAGE IN GENERIC LIBRARIES

In this chapter we present an empirical study of C++ generic libraries with regards to their usage of the programming idioms surveyed in Chapter 3. This work is an extension of the study conducted in [Holeman 2009] on identifying programming idioms in C++ generic libraries. We have extended this work by adapting it to the notion of micropatterns [Gil, Maman 2005], 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. The reverse engineering tools used to conduct the study are described in Chapter 5. This chapter describes two major contributions to the dissertation: a description of the techniques used for conducting empirical studies of generic libraries and the resulting observations derived from this work. This study also demonstrates the viability of the srcML+srcTools based approach for fact extraction for large-scale empirical studies.

6.1 Micropatterns for 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 [Gil, Maman 2005]. 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 constrained templates, which deserve special attention 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 2. We provide a mapping of each micropattern to the idiom that it represents.

Table 2. 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

<table>
<thead>
<tr>
<th>Degenerate Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Degenerate State and Behavior</strong></td>
</tr>
<tr>
<td>Designator</td>
</tr>
<tr>
<td>Taxonomy</td>
</tr>
<tr>
<td>Joiner</td>
</tr>
<tr>
<td>Traits Class</td>
</tr>
<tr>
<td><strong>Metaprogramming</strong></td>
</tr>
<tr>
<td>Type Metafunction</td>
</tr>
<tr>
<td>Integral Metafunction</td>
</tr>
<tr>
<td><strong>Constant Metafunction</strong></td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
<tr>
<td><strong>Metafunction Class</strong></td>
</tr>
<tr>
<td><strong>Degenerate Behavior</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Degenerate State</strong></td>
</tr>
<tr>
<td><strong>Containment</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Inheritance</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Specialization</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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 to 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 call 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 [Gamma, Helm, Johnson, Vlissides 1995]. In [Duret-Lutz, Géraud, Demaille 2001], 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 Section 3.8. We further note that 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).

### 6.2 Validation

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 2.

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 [Siek, Lee, Lumsdaine 2001], which is part of the Boost C++ Libraries. The BGL is 56 KSLOC [Wheeler 2010], 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 6.1. Counts of these observations are given in Table 3.

### Table 3. Counts of micropattern instances from the Boost Graph Library.

<table>
<thead>
<tr>
<th>Micropattern</th>
<th>Manual</th>
<th>Auto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designator</td>
<td>112</td>
<td>105</td>
</tr>
<tr>
<td>Taxonomy</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Joiner</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Traits Class</td>
<td>58</td>
<td>79</td>
</tr>
<tr>
<td>Type Metafunction</td>
<td>159</td>
<td>111</td>
</tr>
<tr>
<td>Integral Metafunction</td>
<td>73</td>
<td>21</td>
</tr>
<tr>
<td>Constant Metafunction</td>
<td>57</td>
<td>68</td>
</tr>
<tr>
<td>Metafunction Class</td>
<td>39</td>
<td>50</td>
</tr>
<tr>
<td>Functor</td>
<td>176</td>
<td>171</td>
</tr>
<tr>
<td>Function Class</td>
<td>31</td>
<td>32</td>
</tr>
<tr>
<td>Stateless</td>
<td>522</td>
<td>596</td>
</tr>
<tr>
<td>Record</td>
<td>34</td>
<td>28</td>
</tr>
<tr>
<td>Environment</td>
<td>48</td>
<td>51</td>
</tr>
</tbody>
</table>
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. We computed the precision, recall and accuracy for the counts of manually and automatically identified programming idiom instances. For reference, the definitions of these measures are given in Table 4.

**Table 4. Definitions of accuracy, precision, and recall are computed in terms of true/false positives and negatives.**

\[
\text{accuracy} = \frac{tp + tn}{tp + tn + fp + fn}
\]
\[
\text{precision} = \frac{tp}{tp + fp}
\]
\[
\text{recall} = \frac{tp}{tp + fn}
\]

The results of these measures are given in Table 5. 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.

Table 5. Accuracy, precision, and recall for each of the micropatterns observed in the Boost Graph Library.

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

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 crosschecked 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. However, the accuracy, precision, and recall are sufficient to engender confidence in the robustness of the approach.

6.3 Empirical Study

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), the Computational Geometry Algorithms Library (CGAL-3.5), 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 statistics and percentages of micropattern identified by srcfacts are shown in Table 6.

Table 6. SLOCs, number of classes, and micropattern count percentages from GCC’s Standard C++ Library, CGAL, and the Boost C++ Libraries.

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

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 [Gil, Maman 2005]. 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 [Siek, Lumsdaine 2000; Zólyomi, Porkoláb 2004]. 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.

6.4 Discussion and Conclusions

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 actually 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 our existing object-oriented
reverse engineering, program comprehension, and software maintenance tools.

In this paper, we present a tool that is capable of automatically identifying the kinds
of idioms being used in the construction of generic libraries. In a sense, this is not
equally different from the use of stereotypes in UML. The abstract labeling or
stereotyping of elements is often used to improve the ways in which developers interact
with source code [Dragan, Collard, Maletic 2006]. 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. We
address this issue in Section Chapter 7, by deriving an architectural description for C++
generic libraries from the empirical observations conducted in this section.
CHAPTER 7

A REFERENCE ARCHITECTURE FOR GENERIC LIBRARIES

This chapter presents and describes a reference architecture for C++ generic libraries. A reference architecture describes the structural elements of a class of software systems and the relationships between those elements. These are also referred to as components and connectors [Shaw, Garlan 1996]. The intent of describing such an artifact is to raise the level of abstraction in the description of generic libraries. When considering programming-in-the-large tasks, it is more important to reason about the impact or relation of library components rather than individual class templates or metafunctions. We validate the appropriateness of the architecture by performing cluster analysis on the architectural components of the Boost C++ Libraries. The primary research contribution of this chapter is the description of the architecture and its validation.

7.1 Architectural Description

From the study conducted in Section Chapter 6, our experience maintaining the Boost Graph Library [Siek, Lee, Lumsdaine 2001], and knowledge of the proposals for ISO standard of the C++0x programming language [Dos Reis, Stroustrup 2006; Gregor et al. 2006; Siek et al. 2005] we derived a reference architecture for generic libraries. The described architecture bridges the idiomatic composition of existing libraries to the concept-based definitions that will eventually be realized [Stroustrup 2009a]. A visual
representation of this architecture is given in Figure 68. We present empirical evidence for the derivation of this architecture in

Figure 68. Client applications instantiate and use generic algorithms, data structures, and adaptors. Concepts describe and constrain the interoperability of algorithms and data types.

Figure 68 depicts our reference architecture for generic libraries and their interactions with client applications. The mechanism by which C++ supports generic programming is predicated upon the use of templates. A generic algorithm or data structure is parameterized over the types of objects or constant values that are used in its definition. Client applications interact with generic libraries primarily through the instantiation and use of generic algorithms and data structures.

Client applications instantiate generic components over user-defined, concrete data types. These instantiations result in new concrete algorithms or data structures. Elements in a generic library can be classified into three categories: conceptual, behavioral, and structural.
Conceptual elements, or *concepts*, describe generic abstractions within a generic library. In C++, concepts are *predicates* that describe requirements on template parameters. In some cases, a client application may be required to interact directly with conceptual elements, although this largely depends on the design of the library. We note that in an abstract sense, concepts might be thought of as ADTs, ML signatures, or Haskell type classes [Bernardy et al. 2008].

Generic algorithms abstract common or useful computational frameworks that can be adapted to the specific needs of the user. Algorithms are instantiated over data types and functions (or functors) defined by the application. Types used as template arguments must *model* the concepts required by the algorithm. A type models a concept if it exposes the required interface and satisfies the required contracts.

Client applications can also instantiate generic data structures and adaptors to create new data types that interoperate with the library’s algorithms. Data structures provided by a generic library *conform* to a concept defined by the library. Adaptor classes are utilities classes that can be used to ease the process of implementing conformant, user-defined data types or to transform the behavior or structure of an existing data type so that it conforms in a different way.

In the remainder of this section, we further define the roles of the conceptual, structural, and behavioral elements of generic libraries, citing specific examples from open source libraries or C++0x proposals to reinforce definitions.
7.1.1 Conceptual Elements

Concepts play a dual role of expressing abstractions and defining constraints. Unfortunately, the C++ programming language does not yet provide linguistic support for declaring or using concepts, despite a number of viable proposals. As a result, concepts are used either in a documentary form or emulated via programming idioms. The proposed concept syntax and associated programming idioms are described Chapter 2 and Chapter 3.

7.1.2 Behavioral Elements

A generic algorithm is a computational framework that is parameterized over interoperable types and/or functions. Algorithms express constraints on those types using concepts (or concept-checking libraries). Consider the algorithm in Figure 69, adjacent_find which is provided by the C++ Standard Library. The algorithm finds the first element that is equivalent to its successor. This implementation is given using the proposed C++0x concepts syntax.

```cpp
template <Forward_Iterator Iter>
requires Equality_Comparable<
    typename Forward_Iterator<Iter>::reference>
Iter adjacent_find(Iter f, Iter l) {
    if(f == l) return l;
    Iter x = f;
    while(++x != l) {
        if(*f == *x) return f;
        f = x;
    }
    return l;
}
```

Figure 69. The adjacent_find algorithm returns requires the type Iter to be a Forward_Iterator and that its associated value_type be Equality_Comparable.
Here, the `adjacent_find` algorithm requires a) that the iterator types over which it is instantiated are models of `Forward_Iterator` and b) that the dereferenced type of those iterator can be compared via the equality operator.

Generic algorithms are not always simply defined by function templates. From our experience working in this domain, we have identified three kinds of generic algorithms that can be differentiated on their use of function overloading. A brief overview of these differences is given in Table 7.

**Table 7. Kinds of generic algorithms differentiated by their use of overloading.**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>No overloads</td>
</tr>
<tr>
<td>Specialized</td>
<td>Overloaded on types</td>
</tr>
<tr>
<td>Restricted</td>
<td>Overloaded on constraints</td>
</tr>
</tbody>
</table>

Note that C++0x concepts syntax would extend the notion of overloading to allow functions to be differentiated by their use of constraints in addition to the number and types of function arguments.

A generic algorithm that is fully defined by a single function template is an *abstract operation*. This is to say that the algorithm is fully polymorphic with respect to its generic (type) parameters, even though they may be constrained. Under this scheme, we would consider `adjacent_find`, in Figure 69 to be an abstract operation.

A *specialized operation* relies on the standard function overloading mechanics to define more specific behavior for different types or templates. For example, consider the `swap` algorithm, which consists of a single generic (and canonical) implementation shown in Figure 70.
template<typename T>
void swap(T& x, T& y) {
    T z = x;
    x = y;
    y = z;
}

template<typename T>
void swap(List<T>& x, List<T>& y) {
    x.swap(y);
}

Figure 70. The generic swap algorithm exchanges the value of two objects. A specialization for list<T> optimizes the general case.

Note that swap is necessarily inefficient for dynamically sized containers such as vectors, linked lists, trees, hash tables, etc. The algorithm can be extended to include efficient implementations for these data types. Every generic container in the C++ Standard Library provides as function template overload of swap algorithm that delegates to its constant time, member-swap implementation. Such an overload also depicted in Figure 70.

A restricted operation is an algorithm that is constrained by the properties of its types. In modern C++, this is typically done by applying the enable_if to cause substitution failures based on some type predication. In future versions of C++, this will be expressed via a requires clause. A classic example is given in Figure 71. Here, the vector’s range constructor is restricted by the kinds of iterators that it can accept.
Figure 71. The `vector` range constructors are restricted by the kinds of iterators over which they are instantiated.

In this example, both constructors are parameterized over the same iterator types and function parameters: a pair of iterators denoting a range. However, the behavior of the algorithm depends on whether or not the kind of `Iter` is an input iterator (e.g., an input stream iterator) or forward iterator (e.g., a linked list iterator). If no techniques were available to restrict the instantiation of the operation, the compiler could generate incorrect code that resulted in a runtime error. We note that this syntax can be effectively emulated using `enable_if` to define a third and default function parameter.

The number of features available to affect function overloading techniques provides a rich language for defining generic algorithms. As with the study on usage patterns of functions overloads presented in [Wang, Hou 2008], an extended analysis of overload techniques for generic libraries could yield interesting results.

7.1.3 Structural Elements

A generic library may provide generic data structures as an accompaniment to its set of generic algorithms. Generic data structures are typically class templates intended to
facilitate interoperability with a library’s generic algorithms either by their conformance to the library’s concepts or their provision of interfaces that conform to those concepts. Note that we use the word, “conform” instead of model because generic data structures are templates, not types.

The STL provides a number of data structures that encapsulate common ownership scenarios: arrays, vectors, lists, search trees, and hash tables. These generic data structures expose an interface to a range of iterators that can be used to the STL algorithms. Consider a partial listing of a trivialized implementation of the STL vector.

Here, the vector class exposes an iterator interface by defining a) the types of its iterators (pointers), and the b) methods to access the beginning and end of the owned sequence of elements. By abstracting pointers into its contiguous memory structure, the vector class can interoperate with nearly every algorithm defined in the C++ Standard Library.

```cpp
template<typename T>
    requires Object_Type<T>
class vector {
public:
    typedef T* iterator;
    iterator begin() { return start; }
    iterator end() { return finish; }
private:
    T *start, *finish;
};
```

**Figure 72. A partial listing of the vector class template provides access to a range of iterators (pointers).**

By providing generic data structures, a generic library provides a set of class templates that are ready to use with its generic algorithms. As previously stated, it is not
required for a generic library to provide a data structure interface, but it is nearly always done for convenience.

Like generic data structures, generic libraries may also provide *adaptors*, which are utility classes (usually templates) that can be used to generate concept-conformant interfaces, typically as some kind of wrapper class. Client applications instantiate adaptors over data types to create models of concepts that can be used to interoperate with the generic algorithms of a library. For example, the C++ Standard Library provides a number of *iterator adaptors*, class templates that adapt or modify the behavior of traversal and data access operations. Consider a listing of the `reverse_iterator` adaptor, which is shown in Figure 73.

```cpp
template<typename Iter>
    requires Bidirectional_Iter<Iter>
struct reverse_iterator {
    reverse_iterator(Iter x) : base(x) { }
    reverse_iterator& operator++() {
        --base; return *this;
    }
    reverse_iterator& operator--() {
        ++base; return *this;
    }
    typename Iter::reference operator*() {
        return *prev(base);
    }
    Iter base;
};
```

**Figure 73.** The `reverse_iterator` adaptor reverses the traversal direction of any iterator that models the `Bidirectional_Iterator` concept.

The `reverse_iterator` class is parameterized over another application-specific iterator type (possibly taken from a container class) and instantiates a *new* iterator type that reverses the traversal operations of the underlying iterator. This allows us to recombine previous algorithms with the new iterator features to create a new algorithm.
Here, the `adjacent_find` can be used in conjunction with reverse iterators to implement, what would amount to a new algorithm, `adjacent_find_backward`.

### 7.2 Validation

In order to support our observations on the architecture of generic libraries, we used the data from our previous studies to cluster libraries by their architectural composition (number of conceptual, structural, and behavioral elements). If the results of the cluster analysis create groups of libraries that are known to be similar in design and intent, then we can be confident that the proposed reference architecture is appropriate.

To conduct this experiment, we reduced the dimensionality of the micropattern analysis by relabeling the studied classes according to their identified idioms.

- **Conceptual** – any class previously labeled as a type trait (metafunction), a traits class, or a tag class (designator, taxonomy, or joiner) was labeled as a conceptual element.
- **Behavioral** – every functor and free function is counted as a behavioral element.
- **Structural** – any class not previously labeled is labeled as a structural element.

Recall, from Table 1, that there is a close correspondence between a number of programming idioms (i.e., type traits, traits classes, and tag classes) and the proposed C++0x concepts.

In total, we surveyed 66 Boost C++ Libraries. This study only includes libraries encapsulated as top-level include directories within the Boost project (e.g., Iterator and Range). The study explicitly excludes the Preprocessor and Config libraries since they are largely preprocessor-based. We also exclude the pending and detail directories since they do not represent cohesive libraries, but rather collections of relatively
unrelated data structures, algorithms, and metaprogramming utilities. We did not extend the survey to CGAL, GCC Standard Library, or the Boost TR1 library for similar reasons (i.e., too broad in scope). For each library, we computed the percentage of composition for each library: the number of instances of each element divided by total number of elements. A complete listing of these observations is given in Table 8.

**Table 8. The conceptual, structural, and behavioral composition of 66 Boost C++ Libraries by percentage.**

<table>
<thead>
<tr>
<th>Library</th>
<th>Conceptual</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulators</td>
<td>0.33</td>
<td>0.40</td>
<td>0.27</td>
</tr>
<tr>
<td>algorithm</td>
<td>0.08</td>
<td>0.05</td>
<td>0.87</td>
</tr>
<tr>
<td>archive</td>
<td>0.03</td>
<td>0.51</td>
<td>0.45</td>
</tr>
<tr>
<td>asio</td>
<td>0.04</td>
<td>0.29</td>
<td>0.67</td>
</tr>
<tr>
<td>assign</td>
<td>0.06</td>
<td>0.12</td>
<td>0.82</td>
</tr>
<tr>
<td>bimap</td>
<td>0.38</td>
<td>0.26</td>
<td>0.36</td>
</tr>
<tr>
<td>bind</td>
<td>0.07</td>
<td>0.09</td>
<td>0.84</td>
</tr>
<tr>
<td>circular_buffer</td>
<td>0.04</td>
<td>0.11</td>
<td>0.85</td>
</tr>
<tr>
<td>concept_check</td>
<td>0.00</td>
<td>0.63</td>
<td>0.38</td>
</tr>
<tr>
<td>concept</td>
<td>0.36</td>
<td>0.55</td>
<td>0.09</td>
</tr>
<tr>
<td>date_time</td>
<td>0.10</td>
<td>0.19</td>
<td>0.71</td>
</tr>
<tr>
<td>dynamic_bitset</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>exception</td>
<td>0.13</td>
<td>0.29</td>
<td>0.59</td>
</tr>
<tr>
<td>filesystem</td>
<td>0.01</td>
<td>0.08</td>
<td>0.91</td>
</tr>
<tr>
<td>flyweight</td>
<td>0.42</td>
<td>0.44</td>
<td>0.14</td>
</tr>
<tr>
<td>format</td>
<td>0.07</td>
<td>0.21</td>
<td>0.72</td>
</tr>
<tr>
<td>functional</td>
<td>0.03</td>
<td>0.16</td>
<td>0.81</td>
</tr>
<tr>
<td>function</td>
<td>0.24</td>
<td>0.33</td>
<td>0.43</td>
</tr>
<tr>
<td>function_types</td>
<td>0.52</td>
<td>0.47</td>
<td>0.01</td>
</tr>
<tr>
<td>fusion</td>
<td>0.43</td>
<td>0.35</td>
<td>0.22</td>
</tr>
<tr>
<td>gil</td>
<td>0.01</td>
<td>0.59</td>
<td>0.40</td>
</tr>
<tr>
<td>graph</td>
<td>0.19</td>
<td>0.17</td>
<td>0.65</td>
</tr>
<tr>
<td>Category</td>
<td>0.25</td>
<td>0.75</td>
<td>0.00</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>interprocess</td>
<td>0.13</td>
<td>0.22</td>
<td>0.65</td>
</tr>
<tr>
<td>intrusive</td>
<td>0.24</td>
<td>0.20</td>
<td>0.56</td>
</tr>
<tr>
<td>io</td>
<td>0.50</td>
<td>0.50</td>
<td>0.00</td>
</tr>
<tr>
<td>iostreams</td>
<td>0.25</td>
<td>0.32</td>
<td>0.42</td>
</tr>
<tr>
<td>iterator</td>
<td>0.27</td>
<td>0.44</td>
<td>0.29</td>
</tr>
<tr>
<td>lambda</td>
<td>0.47</td>
<td>0.15</td>
<td>0.38</td>
</tr>
<tr>
<td>logic</td>
<td>0.00</td>
<td>0.10</td>
<td>0.90</td>
</tr>
<tr>
<td>math</td>
<td>0.04</td>
<td>0.04</td>
<td>0.92</td>
</tr>
<tr>
<td>mpi</td>
<td>0.04</td>
<td>0.26</td>
<td>0.70</td>
</tr>
<tr>
<td>mpl</td>
<td>0.72</td>
<td>0.26</td>
<td>0.02</td>
</tr>
<tr>
<td>multi_array</td>
<td>0.36</td>
<td>0.26</td>
<td>0.38</td>
</tr>
<tr>
<td>multi_index</td>
<td>0.12</td>
<td>0.23</td>
<td>0.65</td>
</tr>
<tr>
<td>numeric</td>
<td>0.13</td>
<td>0.13</td>
<td>0.74</td>
</tr>
<tr>
<td>optional</td>
<td>0.06</td>
<td>0.08</td>
<td>0.86</td>
</tr>
<tr>
<td>parameter</td>
<td>0.52</td>
<td>0.27</td>
<td>0.21</td>
</tr>
<tr>
<td>pool</td>
<td>0.21</td>
<td>0.71</td>
<td>0.07</td>
</tr>
<tr>
<td>program_options</td>
<td>0.00</td>
<td>0.37</td>
<td>0.63</td>
</tr>
<tr>
<td>property_map</td>
<td>0.15</td>
<td>0.24</td>
<td>0.61</td>
</tr>
<tr>
<td>property_tree</td>
<td>0.04</td>
<td>0.22</td>
<td>0.73</td>
</tr>
<tr>
<td>proto</td>
<td>0.42</td>
<td>0.40</td>
<td>0.18</td>
</tr>
<tr>
<td>ptr_container</td>
<td>0.14</td>
<td>0.20</td>
<td>0.66</td>
</tr>
<tr>
<td>python</td>
<td>0.13</td>
<td>0.41</td>
<td>0.47</td>
</tr>
<tr>
<td>random</td>
<td>0.27</td>
<td>0.17</td>
<td>0.56</td>
</tr>
<tr>
<td>range</td>
<td>0.44</td>
<td>0.21</td>
<td>0.35</td>
</tr>
<tr>
<td>regex</td>
<td>0.04</td>
<td>0.14</td>
<td>0.83</td>
</tr>
<tr>
<td>serialization</td>
<td>0.10</td>
<td>0.29</td>
<td>0.61</td>
</tr>
<tr>
<td>signals2</td>
<td>0.11</td>
<td>0.30</td>
<td>0.59</td>
</tr>
<tr>
<td>signals</td>
<td>0.16</td>
<td>0.41</td>
<td>0.43</td>
</tr>
<tr>
<td>smart_ptr</td>
<td>0.07</td>
<td>0.20</td>
<td>0.74</td>
</tr>
<tr>
<td>spirit</td>
<td>0.23</td>
<td>0.35</td>
<td>0.42</td>
</tr>
<tr>
<td>statechart</td>
<td>0.09</td>
<td>0.38</td>
<td>0.53</td>
</tr>
<tr>
<td>system</td>
<td>0.19</td>
<td>0.15</td>
<td>0.67</td>
</tr>
<tr>
<td>test</td>
<td>0.05</td>
<td>0.27</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>0.22</td>
<td>0.75</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>thread</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tuple</td>
<td>0.16</td>
<td>0.19</td>
<td>0.65</td>
</tr>
<tr>
<td>typeof</td>
<td>0.87</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>type_traits</td>
<td>0.33</td>
<td>0.55</td>
<td>0.12</td>
</tr>
<tr>
<td>units</td>
<td>0.37</td>
<td>0.36</td>
<td>0.27</td>
</tr>
<tr>
<td>unordered</td>
<td>0.12</td>
<td>0.26</td>
<td>0.63</td>
</tr>
<tr>
<td>uuid</td>
<td>0.05</td>
<td>0.05</td>
<td>0.89</td>
</tr>
<tr>
<td>variant</td>
<td>0.33</td>
<td>0.42</td>
<td>0.25</td>
</tr>
<tr>
<td>wave</td>
<td>0.08</td>
<td>0.16</td>
<td>0.76</td>
</tr>
<tr>
<td>xpressive</td>
<td>0.15</td>
<td>0.30</td>
<td>0.55</td>
</tr>
</tbody>
</table>

From this data, we can interpret each of the Boost libraries as a 3-vector representing their architectural composition and perform cluster analysis to deduce categorizations. Specifically, we use cluster each library using cosine similarity with an agglomerative hierarchical method and flexible linkage with a $\beta$-value of $-0.25$, which prevents overly flexible linking of clusters. Although we experimented with a number of clustering methods (simple and complete linkage), the results generated using this particular set of parameters generated the most visually representative clustering. We do note, however, that all experimental clustering results contained partitions for data structures and algorithms and concept-oriented or metaprogramming libraries. The resulting dendrogram is shown in Figure 74. The clustering threshold (0.32) indicates that there are three top-level clusters of libraries that are evidentially distinguished by the extent of the architectural elements representing each library. We have labeled several clusters based on the similarity of features.
Figure 74. A clustering of Boost C++ Libraries by (cosine) similarities, using flexible linking, and based on their composition of architectural elements. The threshold marker (0.32) denotes top-level clusters of in terms of architectural composition. Clusters are labeled for convenience.

The leftmost, top-level cluster is comprised of elements that are largely conceptual and contains a sub-cluster of libraries that include a substantial metaprogramming component. For example, the Type_traits, Function_types, concept, and MPL libraries are comprised of utilities for emulating C++ concepts. Libraries such as Bimap, Lambda, Proto, Fusion, Variant, and Units rely heavily on metaprogramming to compose type abstractions.

The rightmost clusters represent basic branches of abstraction and algorithm libraries. The abstraction cluster is comprised of libraries that provide data abstractions (e.g., Intrusive, Multi_index, Unordered, Graph, and Tuple), and system or language
abstractions (e.g., Thread, Filesystem, Asio, Date_time, Regex, Functional, and Bind). The algorithm cluster contains libraries that are largely comprised of functional or procedural abstractions. Examples include Archive, GIL, Function, Python, and Signals.

The clusters emerging from this analysis lends credence to architectural description in terms of conceptual, structural, and behavioral elements. Had the resulting analysis not included these components, we would be forced to conclude that the architecture did not adequately describe the components of these libraries. This not being the case, we can conclude that reference architecture is appropriate for the kinds of libraries surveyed.

We note that there are some anomalies in cluster analysis that can be traced back to markup or parsing errors in reverse engineering tools used to identify these idioms. We would expect, for example the Pool library (which includes a number data structures for object or allocation pools) to appear in the abstractions cluster rather than the conceptual cluster. These errors are known and have been documented in our previous studies.

7.3 Conclusions

In this chapter we have an empirically derived basis for understanding of how generic libraries are constructed from programming idioms, and how those idioms are reflected as architectural elements in the libraries’ organizations. We have also described the tools and source code models used to reverse engineer, decompose, and analyze these libraries. The work presented in this chapter provides a foundation for future work in the development of tools and techniques supporting design recovery and program comprehension. Moreover, by elevating the level of abstraction used to describe generic libraries, we can more accurately reason about the impact of maintenance and evolution.
on libraries and their client applications; the work directly supports the notion of “generic programming-in-the-large”. We leverage the architectural knowledge presented in this chapter to describe a new technique for maintaining and evolving C++ templates in Chapter 8.
CHAPTER 8

MIGRATING TEMPLATES TO CONCEPTS

In this chapter, we present an approach to support the evolution of modern C++
generic libraries to use C++0x concepts. This work was presented in [Sutton, Maletic
2008], before the removal of concepts from the C++0x programming language
[Stroustrup 2009a]. Evidence presented in Chapter 3 indicates that concepts are so
fundamental to the organization and implementation of generic libraries that their
removal is, at worst, temporary. Unfortunately, a future revision of concepts may not
reflect the syntax or models targeted in this chapter. However, the underlying technique
is sound and should scale, in part, to any subsequent proposals.

This work presented in this chapter contains a number of research contributions of
the overall dissertation. Specifically, this chapter addresses a new problem in software
maintenance and evolution with regards to generic programming and generic libraries.
We describe (in Section 8.2) and validate (in Section 8.3) a new source code analysis
technique for identifying template constraints in function templates. Development and
experimentation with this technique has led to a number of interesting observations about
the constraint of generic algorithms and the design of concept hierarchies. Results from
this work were communicated to the C++ standards committee and were used to resolve
several issues with the specification of constraints and concepts in the Standard Library.
This is discussed in Section 8.4.
8.1 Overview

The primary goal of using concepts with generic algorithms is to constrain the set of types over which a template can be instantiated. However, in an unconstrained template, this begs the question of which constraints are the most appropriate. Although seemingly simple to address, the specification of constraints is a careful balancing act between generality and safety. Said otherwise, we do not constraints to exclude viable template arguments, yet we need to ensure that the constraints are not so permissive as to allow for invalid instantiations, which might cause errors manifested at runtime. Consider a simple implementation of the `fill` algorithm, shown in Figure 75.

```cpp
template<typename Iter, typename T>
void fill(Iter f, Iter l, T const& x) {
    for( ; f != l; ++f) {
        *f = x;
    }
}
```

**Figure 75. A simple implementation of the generic `std::fill` algorithm.**

Experienced developers should immediately recognize the function template can be instantiated by most sequences providing an iterator (e.g., `Vector`) and pointers into arrays (e.g., `int*`). Likewise, an experienced developer would be able to tell you that `int` is not a viable substitution since integer types are not dereferenceable (i.e., the expression `*f` will result in a compiler error). However, there are a number of subtleties in this template that are not immediately obvious. For example, this algorithm cannot be instantiated over the iterators standard pair-associative containers (e.g., `Map`) due to the assignment in the expression `*f = x`. If `*f` results in a key-value pair of a `std::map`,
the assignment will give a compiler error because the key component is `const-qualified` (i.e., is immutable).

Constraining the template parameters with concepts provides a method for the compiler to catch and diagnose template parameter substitution errors before instantiation and lookup errors. Using concepts above, we might consider redefinition of the `fill` algorithm as shown in Figure 76.

```cpp
template <typename Iter, typename T>
requires Arithmetic_Like<Iter>
    && Has_Dereference<Iter>
    && Assign<Has_Dereference<Iter>::result_type, T>>
void fill(Iter f, Iter l, T const& x) {
    for( ; f != l; ++f) {
        *f = x;
    }
}
```

**Figure 76.** A new definition for the `fill` algorithm that lists requires the satisfaction of concept instances in order to be instantiated.

The choice of requirements used to constrain the template is largely up to the programmer. The C++0x concepts proposal admits the possibility of many different concepts expressing requirements on the same data template parameter, and the only requirement for constraints is that they adequately cover the set of expressions within the function template. Although this sequence of requirements does cover the expressions of the `fill` algorithm, it fails to capture the semantics of the types involved. The intent of the `Iter` parameter is that it represents an iterator and not simply an arithmetic-like type that also happens to be dereferenceable. While this requirements clause fails to capture the true semantics of the template parameters in this function, it is not technically incorrect.
We note that any method of deducing constraints on a template should attempt to distinguish syntactically and semantically correct requirements, as illuminated by the discussion above. In the Section 8.2, we describe our approach to automatically identifying constraints on function templates.

8.2 Approach

Our approach to the automatic identification of requirements begins at the source code level, extracting operations and types (requirements) associated with expressions involving template parameters. Subsequent analyses match these requirements to concepts, refine the candidates, and finally compute the suggested data abstractions (concept instance) for each template parameter. The algorithm takes as input the AST of a function template, and an ASG containing a set of previously defined concepts (called the concept repository) as represented by the syntax presented in Section 2.2. The output of the algorithm is a set of constraints representing alternative specifications of requirements for the input function template. The specific stages of the approach are:

- **Requirement extraction** – The function template is analyzed for requirements, producing a list of requirements.

- **Concept matching** – Extracted requirements are matched against a set of concept definitions to find those that express the given requirements.

- **Concept instantiation** – Matched concepts are instantiated by correlating types in the extracted requirements to those in the concept definitions.

- **Refinement search** – The refinement hierarchy is instantiated and searched to find additional candidate concept instances.
• **Formal concept analysis** – A binary relation is constructed over the set of concept instances and their requirements and is used to generate a partial ordering of concept instances and their requirements. Note the term “concept” in formal concept analysis is unrelated to the term “concept” in C++0x concepts. This is simply a collision of terminology.

The concept instances that represent the data abstractions of the template parameters to this function are extracted from this lattice. The following sections explain each stage in detail.

### 8.2.1 Requirement Extraction

The first phase of the concept identification algorithm starts with the bottom-up task of extracting requirements within the source algorithm. Much like well-known type inference algorithms [Milner 1978], we traverse the AST of the function template in order to deduce an assignment of types to expressions. However, when the traversal encounters an expression involving operands whose types involve template parameters, we formulate a type requirement for the expression, and propagate a placeholder for the resulting expression type.

We define a type requirement as a function signature formulated over the types or placeholders in the expression. The names of functions and types are encoded during this phase to reduce the possibility of ambiguities (e.g., unary/binary operators, free/member functions). Note that these function declarations only include a return type when the return is required by another expression. Consider, from the **fill** algorithm, the expression, `f != l`. Because this expression is in the condition clause of the **for** loop,
we can anticipate that its return type is `bool`. The entire expression results in the type requirement `neq(Iter, Iter) -> bool`. Conversely the expression `++f` results in the requirement `preinc(Iter)`, with no return type.

In many cases, the types present in an expression depend on the result types of other expressions. Since we cannot generally deduce the return type of an expression, the types used in the formulation must be symbolically written. The expression `*f = x` is an example of type dependency between statements, consisting of dereferencing `f` and assigning the value `x` to that result. The resulting requirements are `operator*(Iter) -> *Iter` and `assign(*Iter, T)`. Here, the left-hand type is depends on the result of the dereference operator, and is generated by prefixing the operator with the type involved, creating a unique type name for result. Other dependent requirements can be similarly derived. A complete table of type requirements for the fill algorithm is given in Table 9.

**Table 9. Expressions and their associated requirements extracted from the fill algorithm.**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter f, l</td>
<td>Iter::ctor(Iter)</td>
</tr>
<tr>
<td>f != l</td>
<td>neq(Iter, Iter) -&gt; bool</td>
</tr>
<tr>
<td>++f</td>
<td>preinc(Iter)</td>
</tr>
<tr>
<td>*f</td>
<td>deref(Iter) -&gt; *Iter</td>
</tr>
<tr>
<td>*f = x</td>
<td>assign(*Iter, T)</td>
</tr>
</tbody>
</table>

### 8.2.2 Concept Matching

Having extracted requirements, we search the concept repository to find a set of concept definitions that express the same requirements. The concept repository is an
ASG containing a collection of concept definitions, their refinements, associated functions, types and requirements. The model of the ASG used in this analysis is the same as that described in Section 5.2. The concept repository contains the concept definitions proposed for the C++ Standard Library, but could conceivably contain concept definitions for any domain (e.g., graph data structures and algorithms).

We begin by preprocessing the repository. For each definition in the repository, we instantiate associated (nested) requirements and add any resulting function definitions as members of the nesting concept definition. This has the effect of inserting related operations on associated types into the scope of these concept definitions. Also, each operator, constructor and destructor is also rewritten using the same symbolic using the approach given above. The results of this preprocessing on the `OutIter` concept are shown in Figure 77.

```plaintext
concept Output_Iterator<X> : Iterator<X> {
    X& preinc(X&);
    reference deref(X&);
    reference& assign(reference&, const value_type&);
};
```

**Figure 77. A partial listing of the preprocessed **OutIter** concept definition.**

Finally, a mapping is created that associates the name of each concept member (functions and types) with the set of concepts that include them. Given this mapping it is now a trivial task to match the type requirements identified in Table 9 to a corresponding set of concepts. The results of this initial mapping is shown in Table 10.

**Table 10. The initial set of concepts that match the type requirements extracted from the fill algorithm.**

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The mapping from type requirements to concept definitions is then inverted such that each concept is now mapped to the set of type requirements that it describes and their corresponding set of concept members. This allows us to work with only the relevant members of each concept with respect to the original function. The results of this initial matching for the fill algorithm are given in Table 11.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy_Constructible</td>
<td>Iter::ctor(Iter)</td>
</tr>
<tr>
<td>Equality_Comparable</td>
<td>neq(Iter, Iter)-&gt;bool</td>
</tr>
<tr>
<td>Has_Dereference</td>
<td>deref(Iter)-&gt;*Iter</td>
</tr>
<tr>
<td>Arithmetic_Like</td>
<td>preinc(Iter)</td>
</tr>
<tr>
<td>Iterator</td>
<td>preinc(Iter)</td>
</tr>
<tr>
<td>Output_Iterator</td>
<td>preinc(Iter)</td>
</tr>
<tr>
<td>Has_Assign</td>
<td>assign(*Iter, T)</td>
</tr>
</tbody>
</table>

It is important to realize that the concepts in Table 11 can be mapped to requirements on different types. This problem will be resolved during concept instantiation.
8.2.3 Concept Instantiation

Concept instantiation is similar to template instantiation in that concrete types are substituted for concept parameters, resulting in a mechanism that can be used to validate or specialize the concrete types within the context of a function template. Unlike traditional instantiation processes, which might be described as “top-down”, our approach requires us to implement “bottom-up” instantiation. We cannot be sure which template parameters are going to being used as arguments to candidate concepts, or whether or not the concept instance can be generated correctly.

The inputs to the bottom-up instantiation algorithm are a concept definition and a set of type requirements. From these inputs, we must deduce a) the number of distinct concept instances represented by the set of type requirements and b) the types that will act as arguments for concept parameters. A general outline of the algorithm follows:

Analyze each concept identified in the previous stage in order. For each concept, sort its associated set of requirements. This orders requirements with no computed types before those that return computed types before those that take computed types as arguments. For example, the order of requirements for the Output_Iterator concept is preinc, deref, and assign. This ordering helps iteratively construct instances by reducing the probability that a computed type (e.g., *Iter) will not be associated with a type given in the concept member (e.g., reference).

We can now instantiate a set of concept instances with respect to the current concept definition and its set of type requirements. Create an empty set of pseudo-concepts $X$. A pseudo-concept is like a scratch pad that stores additional information and can be used to
instantiate a concept definition. Specifically these structures store mappings between types in the concept definition and those in the type requirements. For each type requirement, create a new pseudo-concept \( x \). Identify the concept member that corresponds to the current requirement (i.e., has the same name). Perform a pairwise comparison of the types in the requirement (call these types set \( S \)) with those present in the concept member (call types these set \( T \)). If a type \( s \in S \) is compatible with a type \( t \in T \), record the mapping of \( t \) to \( s \) in the pseudo-concept. If the pseudo-concept is not in \( X \), add it. If it is, merge the mapping of types \( t \) to \( s \) into the instance \( x \).

If types are computed as the result of an operation (e.g., \(*\text{Iter} \) ), identify the best candidate instance in \( X \) and try to identify a corresponding type. Because of the ordering of requirements, it is more likely that a mapping from the computed type to one in the concept definition will have been recorded (as a return type). If such a mapping exists, no more work need be done.

Type compatibility is essentially determined by the rules of type conversion in C++. For example, a \texttt{const} reference or pointer cannot be implicitly converted to a non-\texttt{const} reference or pointer. Objects and built-in types are not implicitly convertible, etc. If, at any point, a requirement has a type that is not type compatible with one in the concept definition and hence not mapped to anything, the current pseudo-concept is rejected.

After evaluating all requirements, the set \( X \) will contain a set of potential pseudo-concepts for the input function template. A pseudo-concept can be instantiated if each concept parameter in its concept definition is mapped to a type present in one of the type
requirements. If so, the pseudo-concept can be used to generate a requirement over on those types.

To help elucidate this process, we give the following example. Suppose, for example, that we are instantiating the Copy_Constructible concept over its requirement, `Iter::ctor(Iter)`. We start by creating a pseudo-concept for this requirement. We must now compare this requirement to its corresponding requirement defined by the Copy_Constructible concept: `T:ctor(const T&)`. Here, `Iter` is compatible with `T`, so we add it to mapping for the pseudo-concept. The next type `Iter` is also compatible with the parameter `T const&`, but the mapping is already made so no action need be taken. By iterating over the concept parameters in the definition, we find that the only parameter `T` is mapped to `Iter` so we generate the requirement `Copy_Constructible<Iter>`, which we can now use to constraint the original function template.

As this algorithm operates on the intricacies of the C++ program model, it goes without saying there are a number of corner cases that can best be solved heuristically. For example, requirements are not rejected simply because they have a computed or derived type as an argument. Also, arguments to concepts with default or variadic parameters are “telescoped” when applicable. This is to say that if the name of the argument is the same as the argument referred to by the default parameter, then the argument list is truncated on the first such occurrence. For example, if suppose we are instantiating the concept Has_Equal, whose definition is given in Figure 78.
concept Has_Equal<typename T, typename U = T> {
    typename result_type;
    result_type operator==(T const&, U const&);
}

Figure 78. The Has_Equal concept takes two parameters, but the default value of the second is given as the first.

If we are instantiating this concept and have deduced the assignment of both parameters, and the first is the same as the second, then telescoping the arguments (say, Iter) will result in the requirement, Has_Equal<Iter> rather than Has_Equal<Iter, Iter>. Variadic templates are also given special consideration.

Consider the Has_Constructor concept, define in

concept Has_Constructor<typename T, typename... Args> {
    T::T(Args...);
}

Figure 79. The Has_Constructor concept requires a type T to be constructible over a sequence of argument types.

In this case, we match the first deduced argument to T and any remaining arguments to the variadic type parameter, Args. There are several other corner cases that must be accommodated relating to const/volatile qualifiers and referential binding. The final set of concept instances for the fill algorithm is given in Table 12

Table 12. The initial set of requirements extracted for the fill algorithm.

<table>
<thead>
<tr>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy_Constructible&lt;Iter&gt;</td>
</tr>
<tr>
<td>Equality_Comparable&lt;Iter&gt;</td>
</tr>
<tr>
<td>Has_Dereference&lt;Iter&gt;</td>
</tr>
<tr>
<td>Arithmetic_Like&lt;Iter&gt;</td>
</tr>
<tr>
<td>Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>Output_Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>Has.Assign&lt;*Iter, T&gt;</td>
</tr>
</tbody>
</table>
In subsequent stages of the search, we expand upon this initial set of candidates by searching the concept repository’s refinement lattice.

### 8.2.4 Refinement Search

Unfortunately, the best description of a template parameter’s data abstraction may not be identified during concept matching and instantiation. Instead, it may be a refinement of one of the previously identified concept instances. This stage of the concept identification process is to search the refinement hierarchy for additional candidates. The refinement hierarchy is taken directly from the refinement relations in the concept repository referenced earlier. We want to constrain the inclusion of refinements in our search to exclude those that represent “redundant” expressions of required functions and types.

To do so, we consider the reverse graph of the hierarchy defined by refinements in the concept repository such that each edge is directed from the refined concept to the refining concept. Each of the concept instances in the previous stage is used to generate an instantiation of this graph by propagating instantiations along these edges. The result is a set of disjoint directed acyclic graphs that represent the possible refinements of all concept instances identified in the previous stage. Two new vertices, \( top \) and \( bottom \), are added to the graph such that \( top \) is connected to the roots of each disjoint subgraph, and the terminal nodes in each subgraph are connected to \( bottom \). This creates a lattice over the set of concept instances. Figure 80 shows the subset of this lattice containing the iterator concept instantiations for the fill algorithm.
Concept instances that contain non-redundant expressions of requirements are found at the meet (least upper bound) of combinations of the instances identified in the previous stage. All such concept instances are found by computing the meets for the powerset of the original instances. The bottom element is disregarded since it does not actually represent a concept instance.

![Refinement Lattice for Iter-based Concept Instances](image)

**Figure 80.** The refinement lattice for the **Iter**-based concept instances. Concept names are abbreviated, and initially matched concepts are shown in gray.

For example, the `Mutable_Forward_Iter<Iter>` requirement will be included as a candidate because it is a meet of a refinement of the `Iter` and `Output_Iterator` concepts. However, the `Mutable_Bidirectional_Iterator` concept, also a refinement of `Iterator` and `Output_Iterator`, is not a suitable candidate because it is not the least upper bound of any initial instances. Computing the meets of the powerset of inputs yields the final set of candidates:
Table 13. The initial set requirements generated from the refinement lattice search.

<table>
<thead>
<tr>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy_Constructible&lt;Iter&gt;</td>
</tr>
<tr>
<td>Equality_Comparable&lt;Iter&gt;</td>
</tr>
<tr>
<td>Regular&lt;Iter&gt;</td>
</tr>
<tr>
<td>Has_Dereference&lt;Iter&gt;</td>
</tr>
<tr>
<td>Arithmetic_Like&lt;Iter&gt;</td>
</tr>
<tr>
<td>Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>Output_Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>Mutable_Forward_Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>Has_Assign&lt;*Iter, T&gt;</td>
</tr>
</tbody>
</table>

This algorithm has added two new candidates: Regular<Iter> and Mutable_Forward_Iterator<Iter>. The Regular<Iter> instance is added because it is the meet of the Copy_Constructible<Iter> and Equality_Comparable<Iter> instances. Mutable_Forward_Iterator<Iter> is added for the reasons described above.

Note that when the refinement hierarchy is instantiated, we also propagate the requirements expressed by each instance to their refinements. At the end of this stage of the algorithm, each resulting concept instance will contain all of the requirements that it represents within the original function template.

### 8.2.5 Concept Analysis

We now have a set of concept instances that can be used to generate requirement clauses by combining them in various ways. However, we would like to identify the best possible requirement clause that includes concept instances that are neither overly specific nor generic. We rely on formal concept analysis (FCA) to compute a lattice that
a) provides a mechanism for constructing requirement clauses and b) orders them in such a way that the best requirement clauses are ordered before less appropriate ones.

Formal concept analysis is an information analysis technique that is used to show relations between a set of objects and the attributes they may share in common [Eisenbarth, Koschke, Simon 2001; Snelting 1996]. Formally defined, a formal context $C = (O, A, R)$ is defined as a triple, consisting of a set of objects $O$, a set of attributes $A$, and a binary relation $R$ that associates objects in $O$ with their attributes in $A$. A formal concept $c = (\{a_i\}, \{o_i\})$ is defined as the maximal matching between a set of objects $\{o_i\}$ and attributes $\{a_i\}$ such that all objects $o_i$ possess only and all attributes $a_i$. Equivalently, only attributes $a_i$ are possessed by exactly the objects $o_i$. The set $\{o_i\}$ is the extent of the concept and the set $\{a_i\}$ is its intent.

Formal concept analysis computes a partial ordering over formal concepts based on the subset relation of objects and attributes. The top of the lattice (derived from the partial order) contains the set of all objects that possess no attributes, and the bottom formal concept contains all objects that possess all attributes.

In this context, we are using formal concept analysis to provide an ordering of concept instances (requirements), the best of which will express maximally disjoint subsets of type requirements in the original function template. To do so, we let the previously computed sets of concept instances be the set of objects $O$, and the set of all requirements originally extracted from the function template be the set of attributes $A$. The relation $R$ is constructed such that $(o, a) \in R$ if and only if instance $o$ expresses requirement $a$. The relation $R$, computed for the fill algorithm is shown in Table 14.
Table 14. The relation mapping concept instances to requirements of the `fill` algorithm. Concept arguments and types are omitted for brevity.

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy_Constructible</td>
<td>X</td>
</tr>
<tr>
<td>Equality_Comparable</td>
<td>X</td>
</tr>
<tr>
<td>Has_Dereference</td>
<td>X</td>
</tr>
<tr>
<td>Regular</td>
<td>X</td>
</tr>
<tr>
<td>Arithmetic_Like</td>
<td>X</td>
</tr>
<tr>
<td>Iterator</td>
<td>X</td>
</tr>
<tr>
<td>OutputIterator</td>
<td>X</td>
</tr>
<tr>
<td>Mutable_FORWARD_ITERATOR</td>
<td>X</td>
</tr>
<tr>
<td>Has_Assign</td>
<td>X</td>
</tr>
</tbody>
</table>

This relation used to construct a formal concept lattice that relates each formal concept to its super-concepts and sub-concepts. Here, each formal concept will consist of the set of concepts requirements (objects) that express the requirements for each of the dependent expressions (attributes) in the original algorithm. We say that each formal concept contains the set of requirements that covers a subset of dependent expressions. Because of this, we refer to this particular formal concept lattice as a coverage lattice. The coverage lattice for the `fill` algorithm is shown in Figure 81.
Figure 81. The coverage lattice generated for the concept instances and type requirements in the fill algorithm. In this representation, concept names are abbreviated, function names represented symbolically, and same-type constraints are hidden.

The resulting concept lattice provides the means for computing the best requirements for the original function template. Fortunately, the methods that we have employed as inputs to the construction of this lattice result in lattices with some interesting properties.

Because FCA orders lattices with respect to the maximal subsets of object attribution, the extents of formal concepts toward the bottom of the lattice will contain concept instances that express the most requirements within the context of the original function. The net result is that concept instances in these extents will accurately represent the data abstractions implied by the template parameters of the template function.
Consider the lattice in Figure 81, which represents the final analysis of the `fill` algorithm. Recall from Figure 75 that `fill` has two template parameters, but only one with any concrete type requirements, `Iter`. The minimum (bottom) concept in this lattice is the concept containing the `Mutable_Foward_Iterator<Iter>` instance in its extent. This is the best possible expression of requirements for the `Iter` template parameter given the requirements present in the algorithm, and is, in fact, exactly the requirement specified for the `fill` algorithm in the C++0x proposal.

The solution to this problem will not always manifest in the extent of the bottom concept. In fact, this will not be the case for most generic algorithms. In the presence of multiple, distinct data types, the bottom concept will have an empty extent. The immediate super-concepts of the bottom will, however, contain the concept instances that best describe the abstract data types of the different template parameters, and the union of their intents will be the set of all requirements. Said more simply, the solution to this problem will always be found in extent of the bottom concept or those just above it.

More formally, we claim that the solution—the set of concept instances that best describe the template parameters within the scope of a given function—are found in the extent of the least concept of each disjoint, bounded sublattice whose greatest and least elements are immediately less than the top and greater than the bottom, respectively. The proof of this follows from the definition of formal concepts and the means by which they are ordered.

Note that the truly “best” possible solution would be the extent of the least formal concept containing a single concept instance. This will unambiguously identify the
abstract data type of a template parameter within the context of its usage and the
definition of the concept repository. Unfortunately, we cannot guarantee the uniqueness
of such solutions. For example, consider the trivial function `advance` for which moves a
forward iterator some number of positions. Given the standard concept hierarchy, our
approach will be unable to distinguish between the concept requirements
`Arithmetic_Like` or `Mutable_Forward_Iterator`. The former concept is
identified since it explicitly lists the preincrement operator as a requirement, while the
latter is identified because it is a refinement of both `Iterator` and `Output_Iterator`,
and the `Output_Iterator` is selected because it duplicates the constraints of the
`Iterator` concept. In short the constraints on the algorithm cannot be explicitly
distinguished between a numeric concept or one of several kinds of iterator. This is
largely due to the fact that the only requirements on the `Iter` parameter involve traversal
(i.e., `operator++`, `operator--`, or `operator+=`, depending on which advance
algorithm we consider).

8.3 Evaluation

We have implemented a prototype of our concept identification algorithm as a pipe-
and-filter style tool with two components. The first component, based on srcML and
crcTools [Collard, Kagdi, Maletic 2003], implements the requirement extraction process
and enumerates a listing of type requirements for a function template. The output of this
program is piped to the second, which implements the analyses responsible for concept
identification. The evaluation of our approach demonstrates that the technique is broadly
applicable to a larger number of different algorithms and that the results are stable with
respect to subtle variations of the same algorithm.

8.3.1 Applicability

To test applicability, we ran the algorithm against a selection of generic algorithms
from the STL, including sequence and numeric algorithms. We implemented these
algorithms ourselves in order to eliminate any “interference” from the realities of C++
programming (e.g., the preprocessor, external dependencies, unreadable naming
conventions, etc). We compared the results of these tests against the concept
requirements proposed for their respective algorithms. A selection of results is shown in
Table 15. This selection of algorithms includes a sampling of many of the different
forms of STL algorithms: those that operate on a single sequence, those that operate on
multiple sequences, and those that employ functors on elements of the sequence.

Table 15. A selected listing of STL algorithms, their computed concept requirements
(with abbreviated names). The requirements of each algorithm is described by a
conjunction of concept instances with ambiguities written as disjunctions.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>Mutatile_Forward_Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>find</td>
<td>Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>find_if</td>
<td>Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>find_end</td>
<td>Iterator&lt;Iter1&gt;</td>
</tr>
</tbody>
</table>
For the most part, excepting a fair amount of ambiguity, the requirements identifies for these algorithms align well with their proposed requirements. Unfortunately, the results were not as unambiguous as we had originally hoped. For example, our prototype only identifies unambiguous concepts for iterator types that are dereferenced and assigned values (e.g., the `fill` algorithm). The reasons for this are found in the proposed concept hierarchy. The common iterator operations (increment, dereference, etc.) are multiply specified within this hierarchy: in the input and output iterator concepts. The unfortunate result of this duality is the inevitable inclusion of both as well as their mutual refinement, mutable forward iterator, in the candidate set. Without the requirement on assignment through the reference type (as seen in the `fill` algorithm), our approach cannot disambiguate the actual type of iterator.

Moreover, many of the algorithms for which we have ambiguously identified either input iterators or mutable forward iterators, are actually proposed to require (non-mutable) forward iterators – which our implementation unfortunately fails to list. The cause of this problem is also rooted in definition of the standard concepts. As proposed, the forward iterator is a refinement of an input iterator that admits a simple convertibility requirement (`const X& = x++`). Because it defines no new features as associated functions or types, it is never considered as a candidate for inclusion. However, we
might observe that it sits on the path between the input and mutable forward iterator in the graph and could be included as a viable candidate.

The misidentification of the predicate functors in the `find_if` function is a result of the same artifact. The predicate concept is defined as a refinement of the callable concept whose return type is convertible to `bool`. Unfortunately, this definition is insufficient for our algorithm to incorporate the refined predicate definition into the set of candidates.

However, some of the more interesting requirements being generated are the constructability requirements for the functors’ types. Because functors are passed (and often returned) by value, they must be copy constructible. The current proposals for the STL do not list any constructability requirements on generic algorithms taking a functor as an argument. We believe that this is an error of omission in the proposal.

The final ambiguity is that found in the `accumulate` algorithm (generalized summation). Here, the template parameter `T` is described as requiring either an arithmetic-like type or a random access iterator. However, an instantiation of this algorithm with `T` as a random access iterator is almost certainly a logic error and can only work if the dereferenced type of the `Iter` parameter is the reference type of the random access iterator. This error demonstrates an insufficiency in our approach: the filtering of candidates based on allowable or required conversions between their associated types.

8.3.2 Stability

In order to test stability, we ran the algorithm against small variations of the same algorithms. These variations involved either the expansion of single statements into several, or the compression of many statements into fewer. Two variations of the `fill`
implementation are shown in Figure 82. The first compresses the increment and dereference operations into a single statement. The second expands the assignment to use an explicitly named associated type.

```cpp
template<typename Iter, typename T>
void fill(Iter f, Iter l, const T& x) {
    while(f != l) {
        *f++ = x;
    }
}

template<typename Iter, typename T>
void fill(Iter f, Iter l, const T& x) {
    for( ; f != l; ++f) {
        typename Iter::reference r = *f;
        r = x;
    }
}
```

Figure 82. Two minor variations in the implementation of the `fill` algorithm. The first compacts a number of expressions in the original into a single statement, and the second expands them into multiple statements.

Because these two variations are semantically equivalent to the original (shown in Figure 75), we expect the concept requirements to remain roughly the same. A small sampling of results is shown in Table 16.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>Mutable_Forward_Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>Compact</td>
<td>Mutable_Forward_Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td>Expanded</td>
<td>Iterator&lt;Iter&gt;</td>
</tr>
<tr>
<td></td>
<td>Has_Constructor&lt;reference, reference&gt;</td>
</tr>
<tr>
<td></td>
<td>Has_Assign&lt;reference, T&gt;</td>
</tr>
</tbody>
</table>

Table 16. Concept requirements for minor variations on the same algorithm.

Overall, the stability of our approach across these minor variations is fairly good. Most of the differences that we see in the results have more to do with implicit
conversions between types rather than differences in the primary abstractions of the template parameters. In the cases shown here, the analysis fails to correctly associate the associated reference type with the return value of the dereference iterator and can be attributed to the explicit declaration of the associated type. Correcting this flaw will cause algorithm to determine that the \texttt{Iter} parameter must model a mutable forward iterator.

Evaluation of other algorithms yields similar results; we often find that minor variations in the implementation result in minor differences in the final results. While some of these variations are the result of implementation issues (as seen before), others occur because the variations restrict or change the requirements in subtle ways. For example, our analysis of find variants caused the \texttt{Equality\_Comparable\<\texttt{*Iter, T}>} requirement to be reduced to \texttt{Equality\_Comparable\<T>}.

8.4 Discussion

We had originally hoped that our approach would be capable of extracting unambiguous requirements for these algorithms. Because these algorithms are part of a standard library, we expected their interfaces (and hence their requirements) to be unambiguous and immutable. Instead, we have shown that the results of automatic identification of concepts is dependent upon the unambiguous definition of concept hierarchies, and that minor variations in the implementation of algorithms can cause differences in the resulting abstractions. Far from disappointing, these results show that automated concept identification has tremendous value to developers building or refactoring code into generic libraries.
The existence of ambiguous results during concept identification may indicate problems in the specification of the concept hierarchy. The ambiguities found during evaluation certainly cause us to seek rationale for some of its design choices. We believe that tools such as this can be used to help developers better understand their concept design choices and ultimately build better generic libraries.

The generation of different requirements based on minor variations in the implementation of algorithms is also rooted in processes of generic programming. The process of *lifting*—iteratively abstracting similar concrete algorithms into generic algorithms—requires the study of different concrete algorithms in order to understand the commonalities that can be exploited to produce a generic version. Our evaluation shows that sampling a variety of implementations is vital to the correct specification of requirements for the algorithm. Failing to sample a sufficient variety of implementations can result in overly strict or loose constraints on the implementation and on the types over which they can be instantiated.
This chapter presents results derived from the development of and experimentation with the constraint identification technique presented in Chapter 8. While evaluating the initial versions of our approach, we identified a number of questionable recurring patterns in the constraints being generated for different algorithms—cases where our analysis agreed with neither the proposals, nor our intuition. We identified the cause of these patterns as certain aspects of the concept hierarchy and classified these patterns as potential design flaws. The diagnosis and analysis of these problems, which are a primary contribution of this dissertation, are presented in this chapter.

9.1 Introduction

After our experiments developing and validating the work in Chapter 8, we contacted the primary author of the concept hierarchy and algorithm specifications for C++0x [Gregor, Marcus, Witt, Lumsdaine 2008; Gregor, Siek, Lumsdaine 2008] to discuss some of the issues we uncovered. Specifically we refer to the copy-constructability of function objects and the ambiguity of input and output iterators in almost all of the functions we analyzed. Both of these artifacts can be traced to the design of the concept hierarchy.

New versions of the C++0x proposals address both of these issues. We do not claim credit for describing the underlying problems associated with these artifacts or the solutions that have been proposed to address them. However, we point out that our
reverse engineering approach to identifying concept requirements has been useful in highlighting potential design and specification flaws, and it has had influence on C++0x specifications. Both of these artifacts can be traced back to aspects of the concept lattice from which the requirements were originally generated.

9.1.1 Frequently Co-Occurring Constraints

While evaluating the approach on a series of algorithms, we noticed that many requirements clauses frequently apply the same constraints to the same parameters. This was especially true in the case of STL algorithms that take a function object (e.g., `find_if`, `for_each`, `accumulate`, etc.). It is a convention of the C++ Standard Library that these function objects are passed by value, requiring copy construction. During our experimentation with earlier versions of the proposals, we discovered that the proposal was not requiring the function objects to be copy constructible (either explicitly or as part of the Callable concept). This raises questions. Is there an implicit requirement on copy constructability, or is the requirement just missing? Should all callable objects also be copy constructible or, is it just the case that these objects are copy constructible?

Initially, we had suspected that Callable types should be Copy_Constructible by refinement, and that the requirement was mistakenly omitted from proposal. However, our initial suppositions were partially incorrect. There is a blanket requirement in the Standard Library’s specifications that all function objects are passed by value. It does state that all Callable objects are also required to be Copy_Constructible.
The requirement was missing from the proposal, but the two concepts are not related in the way we had suspected.

Co-occurring constraints can be derived from the formal concept lattice discussed in Section 8.2.5. Formally, a constraint co-occurs when a single template parameter is constrained by multiple, unambiguous, disjoint bounded sublattices in the computed coverage lattice. For example, all algorithms in the Standard Library parameterized over function types have co-occurring constraints describing the required function signature and copy constructability. Co-occurring constraints that are present in a non-trivial percentage of algorithms in a library are said to be frequently co-occurring. This phenomenon is an indicator for two design flaws in concept hierarchies:

1. If the constrained type of the frequent constraints is a template parameter, then the concept hierarchy does not include a concept definition that represents the frequent co-occurrence of those concepts.
2. If one or more of the constrained types is a computed (associated) type, then it is likely that some concept definition does not fully express the semantic requirements of the abstraction.

Recall that concept definitions emerge from frequent usage. If two (or more) constraints frequently co-occur for a single constrained type in a generic library, then it would be natural to assume that this recurrence can be expressed in terms of a concept definition. In the first case, the Callable and Copy_Constructible concepts appear for all function objects. It would be possible to create a new concept definition called, say Std_Callable, which refines both of concepts. This would be an appropriate
solution because the concept emerges from usage within the Standard Library, but may not be the case elsewhere.

In the second case, we have frequent constraints that describe a relationship between a constrained type and one or more computed (associated) types. For example, consider the `fill` algorithm shown in Figure 83.

```cpp
template <typename Iter, typename T>
requires OutputIterator<Iter>
&& Has_Assign<typename Iter::reference, X>
void fill(Iter f, Iter l, T const& x) {
    for( ; f != l; ++f) *f = x;
}
```

**Figure 83.** A simple implementation of the `fill` algorithm.

If the `OutputIterator` and `Has_Assign` constraints frequently co-occur, then it may be possible to rewrite the `OutputIterator` (the dominant abstraction) to include requirements for assignability. The concept definition of `OutputIterator` could be modified shown in Figure 84. By rewriting the concept in this manner, we can eliminate the co-occurrence of the `Has_Assign` concept. Specifically, we add semantic requirement to the definition that requires the iterator’s `value_type` to be assignable to the result of dereferencing it (i.e., \( *f = x \)).

```cpp
concept OutputIterator<typenamem X> {
    typename value_type;
    typename reference;
    requires Has_Assign<reference, value_type>;
    reference operator*(X const&);
};
```

**Figure 84.** An `OutputIterator` concept definition that expresses a semantic requirement for its reference type to be assignable to its value type.
This will, however, introduce a same-type constraint between \( T \) and the output iterator’s `value_type` (i.e., `Same_Type<value_type, T>`). This new constraint is also frequently co-occurring so the problem is not actually solved.

We note that solving concept design problems this kind type is often non-trivial and may require substantial hierarchy redesigns because they can indicate an initially incorrect domain analysis.

### 9.1.2 Ambiguous Constraints

One of the problems encountered frequently while experimentally validating our approach was that of ambiguous constraints. In almost all of the algorithms that we tested, there was at least one coverage lattice whose minimum element contained multiple constraints in its extent, and these were almost always related to conceptual iterator hierarchy.

For over a decade, the iterator hierarchy has been a fixture of the Standard Library. Early attempts to model this as concept hierarchy resulted in a set of concepts that separated traversal semantics (forward, bidirectional, and random access) and read/write capabilities (input and output). This concept hierarchy is shown in Figure 85.
In this hierarchy, both the Input_Iterator and Output_Iterator concepts define many of the same requirements (e.g., increment, dereference, etc.) and propagate them down the refinement hierarchy. When using this hierarchy with our approach, nearly every iterator was suggested to be either an Input_Iterator or a Mutable_Forward_Iterator. Because the input and output concepts define many of the same requirements, they are always selected during concept matching, and the mutable forward iterator is always selected during the refinement search. When no other operators are used (e.g., decrement or indexing), it becomes impossible to distinguish the kind of iterator based on its usage, and the iterator’s coverage lattice contains both constraints in its minimum extent.

When this phenomenon occurs frequently, it can indicate a design flaw in the concept hierarchy: ambiguously specified concepts. We should point out that this is does not necessarily indicate a design flaw. There are many instances in the current concept hierarchy proposal where requirements are defined in terms of another concept using refinement (e.g., Has_Constructor and Copy_Constructible). These constraints
are essentially interchangeable. This specific problem is differentiated from this case by
the definition of ambiguous semantic requirements in more general (base) concepts and
their refinement and/or redefinition in refined concepts.

This flaw is can be closely related to the second class of problems discussed in the
previous section. Recall that one solution to frequent co-occurrence of constraints
involving multiple or computed constrained types is to rewrite the concept definition of
the dominant abstraction to include a semantic requirement for the co-occurring
constraint(s). In this case, the assignability semantic requirement has been added to the
OutputIterator and used to distinguish all refining kinds of iterators. As with the
problem discussed above, there are probably no simple solutions to this class of problem.

Along with other problems (especially infrequency of use and the ability to express
simpler requirements), the entire iterator hierarchy has been redesigned. The new
hierarchy primarily considers iterators based on their traversal operators (forward,
bidirectional, and random access). The output iterator has been redesigned to provide a
requirement of capability rather than being part of the hierarchy. It is no longer related to
any other iterators through refinement. More specifically, an OutputIterator
requires any kind of Iterator (a very general concept) to be assignable through
dereferencing to some value type. A partial listing of its definition is shown in Figure 86.

```cpp
concept OutputIterator <typename Iter, typename Value> {
    requires Iterator<Iter>
    requires Has_Assign<typename Iterator<Iter>::reference, Value>;
}
```

Figure 86. The new OutputIterator expresses an assignment requirement
between an Iterator’s reference type and a value type.
This solution gracefully avoids the problem of ambiguity in the iterator hierarchy by simply “binding” the two concepts together: iterators and writeability. The immediate impact of this solution is that it extracts an orthogonal concern (writeability) from the traversal-based concern of the iterator hierarchy. We also note that this design change should not result in frequent constraints in mutating algorithms (e.g., fill and sort) since the constraint applies to multiple types (thus avoiding problem 1) and not a computed type (also avoiding problem 2).
The dissertation addresses several practical concerns in the software maintenance and evolution of C++ generic libraries, in particular in the areas of understanding and migration. Although generic libraries offer substantial benefits to programmers in terms of abstraction and efficiency, they can be difficult to read, write, and maintain. One reason for these difficulties is the leakiness of the template abstraction and the idiomatic nature of generic library implementations. The research described in this dissertation provides a solid foundation for advancing techniques to support the design, implementation, maintenance, and evolution of C++ generic libraries.

The research investigations presented in this thesis yield a number of research and development contributions in the areas of program comprehension and software maintenance for C++ generic libraries. Specific research contributions in the domain of program comprehension include the development of methods to study idiom usage in C++ generic libraries, the observations derived from those studies [Sutton, Holeman, Maletic 2010a], and the derivation of reference architecture for generic libraries [Sutton, Holeman, Maletic 2010b]. Contributions to the field of software maintenance for generic libraries includes the definition of a new source code analysis technique that directly supports the migration of unconstrained C++ function templates to use C++0x concepts and the maintenance of those constraints [Sutton, Maletic 2008], and the use of that
technique to experimentally identify and diagnose problems in the specification of constraints and the design of concept hierarchies. Specific development contributions include the creation of the Origin C++0x Libraries (especially the Origin Concepts library), and the srcTools reverse engineering and source code analysis framework.

Beyond these contributions to the software engineering and maintenance of C++ generic libraries, the research and experiments presented in this dissertation have led to a great deal of insight on the design and implementation of concepts as a part of the programming language, and will have a direct impact on its evolution. The development of the Origin Concepts Emulation Library presented in Chapter 4 demonstrates a viable implementation technique underlying many of the proposed concepts features. The approach taken in the design of this library contrasts starkly with the proposed semantics for concepts, which mandate a stricter form of model checking rather than simple predicate evaluation. The empirical studies presented in Chapter 5 can also be reinterpreted as a study of concept usage that would allow language designers to evaluate which language features might be deemed the most critical by the users.

This work has also yielded several insights on the design and implementation of generic libraries and concept hierarchies. The work on concept emulation in Chapter 4 motivates the specific solutions for decoupling requirements between automatically and implicitly checked concepts. Many of these insights and techniques are reified during the experiments conducted in Chapter 8 and described in Chapter 9. This work will be leveraged to create to new development techniques for constrained generic libraries, and
methods for implementing data type taxonomies (e.g., the algebraic concepts presented in [Gottschling 2006]).

The work presented in this dissertation is provides a solid foundation for the further exploration of problems related to the design and implementation of C++ generic libraries. Future work supported by the research presented in this dissertation includes:

- leveraging the results of this work to design a new set of language features for concepts that focus on the specification and evaluation of constraints,
- investigating alternative uses of concepts for software validation [Bagge, Haveraaen 2009; Zalewski, Schupp 2007] and program optimization [Willcock, Lumsdaine, Quinlan 2009] to motivate or extend a new concept language,
- the extension of the constraint identification algorithm to a) work with class templates and member function templates, b) better diagnosis of constraint specification and coverage errors, c) better diagnose conceptual design flaws, and d) support the automated identification of concepts (not just constraints) in template definitions,
- defining a science of generic library design that incorporates guidelines for the effective design of generic data structures, algorithms, and concepts,
- developing a methodology for using empirical studies of software usage to motivate or constrain the evolution of a programming language,
• deriving theories about the coevolution of programming languages and software based on the development and proliferation of programming idioms and techniques, and
• creating or adapting pedagogical methods for teaching programming concepts for generic programming, library design, and the software engineering of generic libraries.

Generic programming and generic libraries will become increasingly important in the software development community, especially in C++ because they provide features that few other paradigms can offer: high quality, consistent abstractions and unsurpassed efficiency. As interest in this style of programming and the number of libraries increase, engineering rigor and process must be put in place to ensure the continuing quality of these libraries over time. The work presented in this dissertation provides initial steps in those directions.
APPENDIX A

CONCEPTS FOR C++0X

This appendix gives the definitions of many of the concepts used or referenced throughout the dissertation. The concepts given here are not necessarily equivalent to the concepts defined in the C++0x concept proposals, and are only intended to be sufficient to support a more thorough understanding of the subject matter. Note that many of the concepts ultimately rely on the following concept definition for True, and its concept map. This allows the evaluation of arbitrary type traits as part of a requires clause.

```cpp
explicit concept True<bool> { };
concept_map True<true> { };
```

A.1 Support Concepts

The following concepts are used to classify the C++0x type system. We have omitted several concepts from the specification such as Standard_Layout_Type since they are not essential to understanding the core types and type constructors.

```cpp
concept Object_Type<typename T> { 
    requires True<is_object<T>::value>;
}

explicit concept Variable_Type<typename T> { };

explicit concept Constexpr_Type<typename T> : explicit Variable_Type<T> { }

concept Value_Type<typename T> : Object_Type<T>, explicit Variable_Type<T> { 
```
explicit concept Class_Type<typename T>
  : Object_Type<T>
{ }

concept Class<typename T>
  : explicit Object_Type<T>
{ 
  requires True<is_class<T>::value>;
}

concept Polymorphic_Class<typename T>
  : Object_Type<T>
{ 
  requires True<is_class<T>::value>;
}

concept Union<typename T>
  : explicit Object_Type<T>, Value_Type<T>
{ 
  requires True<is_union<T>::value>;
}

concept Trivial_Type<typename T>
  : Value_Type<T>
{ 
  requires True<is_trivial<T>::value>;
}

concept Literal_Type<typename T>
  : Value_Type<T>
{ 
  requires True<is_literal_type<T>::value>;
}

concept Scalar_Type<typename T>
  : Trivial_Type<T>, explicit Literal_Type<T>
{ 
  requires True<is_scalar<T>::value>;
}

concept Arithmetic_Type<typename T>
  : Scalar_Type<T>
{ 
  requires True<is_arithmetic<T>::value>;
}
concept Floating_Point_Type<typename T> 
  : Arithmetic_Type<T> 
  { 
    requires True<is_floating_point<T>::value>; 
  }

concept Integral_Type<typename T> 
  : Arithmetic_Type<T>, Constexpr_Type<T> 
  { 
    requires True<is_integral<T>::value>; 
  }

concept Enumeration_Type<typename T> 
  : Constexpr_Type<T> 
  { 
    requires True<is_enum<T>::value>; 
  }

concept Same_Type<typename T, typename U> { 
  requires True<is_same<T, U>::value>; 
}

concept Derived_From<typename T, typename U> { 
  requires True<is_base_of<U, T>::value>; 
}

A.2 Standard Concepts

The following concepts are adapted from the C++0x proposals. This set of concepts is largely used as a basis for building more complex, abstract, or domain-specific concept specifications.

concept Explicitly_Convertible<typename T, typename U> { 
  explicit operator U(T const&); 
}

concept Convertible<typename T, typename U> 
  : Explicitly_Convertible<T, U> 
  { 
    operator U(T const&); 
  }

explicit concept Lvalue_Reference<typename T> 
{ }
template<typename T>
concept_map Lvalue_Reference<T&>
{
}

explicit concept Rvalue_Reference<typename T>
{
}

template<typename T>
concept_map Rvalue_Reference<T&&>
{
}

concept Has_Plus<typename T, typename U> {
  typename result_type;
  result_type operator+(T const&, U const&);
}

concept Has_Minus<typename T, typename U> {
  typename result_type;
  result_type operator-(T const&, U const&);
}

concept Has_Multiply<typename T, typename U> {
  typename result_type;
  result_type operator*(T const&, U const&);
}

concept Has_Divide<typename T, typename U> {
  typename result_type;
  result_type operator/(T const&, U const&);
}

concept Has_Modulus<typename T, typename U> {
  typename result_type;
  result_type operator%(T const&, U const&);
}

concept Has_Unary_Plus<typename T> {
  typename result_type;
  result_type operator+(T const&);
}

concept Has_Negate<typename T> {
  typename result_type;
  result_type operator-(T const&);
}

concept Has_Less<typename T, typename U> {

typedef result_type;
result_type operator <(T const&, U const&);
}

concept Has_Greater<typename T, typename U> {
typename result_type;
result_type operator > (T const&, U const&);
}

concept Has_Less_Equal<typename T, typename U> {
typename result_type;
result_type operator <=(T const&, U const&);
}

concept Has_Greater_Equal<typename T, typename U> {
typename result_type;
result_type operator >=(T const&, U const&);
}

concept Has_Equal_To<typename T, typename U> {
typename result_type;
result_type operator ==(T const&, U const&);
}

concept Has_NotEqual_To<typename T, typename U> {
typename result_type;
result_type operator!=(T const&, U const&);
}

concept Has_Logical_And<typename T, typename U> {
bool operator&& (T const&, Y const&);
}

concept Has_Logical_Or<typename T, typename U> {
bool operator||(T const&, Y const&);
}

concept Has_Logical_Not<typename T, typename U> {
bool operator!(T const&, Y const&);
}

concept Has_Bit_And<typename T, typename U> {
typename result_type;
result_type operator &(T const&, U const&);
}

concept Has_Bit_Or<typename T, typename U> {
typename result_type;
result_type operator|(T const&, U const&);
}

concept Has_Bit_Xor<typename T, typename U> {  
typename result_type;
result_type operator^(T const&, U const&);
}

concept Has_Complement<typename T> {  
typename result_type;
result_type operator~(T const&);
}

concept Has_Left_Shift<typename T, typename U> {  
typename result_type;
result_type operator<<(T const&, U const&);
}

concept Has_Right_Shift<typename T, typename U> {  
typename result_type;
result_type operator>>(T const&, U const&);
}

concept Has_Dereference<typename T> {  
typename result_type;
result_type operator*(T&);
result_type operator*(T&&);
}

concept Has_Address_Of<typename T> {  
typename result_type;
result_type operator&(T&);
}

concept Has_Subscript<typename T, typename U> {  
typename result_type;
result_type operator[](T&&, U const&);
}

concept Callable<typename F, typename... Args> {  
typename result_type;
result_type operator()(F&&, Args...);
}

concept Has_Assign<typename T, typename U> {  
typename result_type;
result_type T::operator=(U);  
}
concept Has_Plus_Assign<typename T, typename U> {
    typename result_type;
    result_type T::operator+=(U);
}

concept Has_Minus_Assign<typename T, typename U> {
    typename result_type;
    result_type T::operator-=(U);
}

concept Has_Multiply_Assign<typename T, typename U> {
    typename result_type;
    result_type T::operator*=(U);
}

concept Has_Divide_Assign<typename T, typename U> {
    typename result_type;
    result_type T::operator/=(U);
}

concept Has_Modulus_Assign<typename T, typename U> {
    typename result_type;
    result_type T::operator%=(U);
}

concept Has_Bit_AndAssign<typename T, typename U> {
    typename result_type;
    result_type T::operator&=(U);
}

concept Has_Bit_Or_Assign<typename T, typename U> {
    typename result_type;
    result_type T::operator|=(U);
}

concept Has_Bit_Xor_Assign<typename T, typename U> {
    typename result_type;
    result_type T::operator^=(U);
}

concept Has_Left_Shift_Assign<typename T, typename U> {
    typename result_type;
    result_type T::operator<<=(U);
}

concept Has_Right_Shift_Assign<typename T, typename U> {
    typename result_type;
}
result_type T::operator>>=(U);
}

concept Has_Preincrement<typename T> {  
typename result_type;
result_type operator++(T&);
}

concept Has_Postincrement<typename T> {  
typename result_type;
result_type operator++(T&, int);
}

concept Has_Predecrement<typename T> {  
typename result_type;
result_type operator--(T&);
}

concept Has_Postdecrement<typename T> {  
typename result_type;
result_type operator--(T&, int);
}

concept Has_Comma<typename T, typename U> {  
typename result_type;
result_type operator,(T const&, U const&);
}

concept Predicate<typename F, typename... Args> : Callable<F, Args const&...> {  
requires Convertible<result_type, bool>;
}

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

axiom Consistency(T a, T b) {  
(a > b) == (b < a);  
(a <= b) == !(b < a);  
(a >= b) == !(a < b);
}

axiom Irreflexivity(T a) {  
}
(a < a) == false;
}

axiom Antisymmetry(T a, T b) {
    (a < b) => ((b < a) == false);
}

axiom Transitivity(T a, T b, T c) {
    (a < b && b < c) => (a < c)
}

axiom Transitivity_Of_Equivalence(T a, T b, T c) {
    (!!(a < b) && !!(b < a) && !!(b < c) && !!(c < b)) =>
    (!!(a < c) && !!(c < a))
}

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

axiom Consistency(T a, T b) {
    (a == b) == !(a != b);
}

axiom Reflexivity(T a) {
    a == a;
}

axiom Symmetry(T a, T b) {
    (a == b) => b == a;
}

axiom Transitivity(T a, T b, T c) {
    (a == b && b == c) => (a == c);
}

concept Strict_Weak_Order<typename F, typename T>
: Predicate<F, T, T>
{
    axiom Irreflexivity(F f, T a) {
        f(a, a) == false;
    }
}
axiom Antisymmetry(F f, T a, T b) {
  f(a, b) => !f(b, a)
}

axiom Transitivity(F f, T a, T b, T c) {
  (f(a, b) && f(b, c)) => f(a, c);
}

axiom Transitivity_Of_Equivalence(F f, T a, T b, T c) {
  (!f(a, b) && !f(b, a) && !f(b, c) && !f(c, b)) =>
   (!f(a, c) && !f(c, a))
}

concept Equivalence_Relation<typename F, typename T> :
  Predicate<F, T, T> {
    axiom Symmetry(F f, T a, T b) {
      f(a, b) => f(b, a);
    }

    axiom Transitivity(F f, T a, T b, T c) {
      (f(a, b) && f(b, c)) => f(a, c);
    }
  }

concept Has_Destroyer<typename T> {
  T::~T();
}

concept Trivially_Destructible<typename T> :
  Has_Destroyer<T> {
    requires True<has_trivialDestructor<T>::value>;
  }

concept Has_Virtual_Destroyer<typename T> :
  Has_Destroyer<T>, Polymorphic_Class<T> {
    requires True<has_virtualDestructor<T>::value>;
  }

concept Has_Constructor<typename T, typename... Args> {
  T::T(Args...);
}

concept Constructible<typename T, typename... Args>
concept Default_Constructible<typename T>  
: Constructible<T>  
{
}

concept Trivially_Default_Constructible<typename T>  
: Default_Constructible<T>  
{
  requires True<has_trivial_default_constructor<T>::value>;
}

concept Move_Constructible<typename T>  
: Constructible<T, T&&>  
{
}

concept Trivially_Move_Constructible<typename T>  
: Move_Constructible<T>  
{
  requires True<has_trivial_move_constructor<T>::value>;
}

concept Copy_Constructible<typename T>  
: Move_Constructible<T>, Constructible<T, T const&>  
{
}

concept Trivially_Copy_Constructible<typename T>  
: Copy_Constructible<T>  
{
  requires True<has_trivial_copy_constructor<T>::value>;
}

concept Move_Assignable<typename T>  
: Has_Assign<T, T&&>  
{
}

concept Copy_Assignable<typename T>  
: Move_Assignable<T>, Has_Assign<T, T const&>  
{
}

concept Trivially_Move_Assignable<typename T>  
: Move_Assignable<T>  
{
  requires True<has_trivial_move_assign<T>::value>;
}

concept Trivially_Copy_Assignable<typename T>
: CopyAssignable<T>
{
    requires True<has_trivial_copy_assign<T>::value>;
}

concept HasSwap<typename T, typename U> {
    void swap(T, U);
}

concept Swappable<typename T> :
    HasSwap<T&, T&>
{
}

concept Semiregular<typename T> :
    Copy_Constructible<T>, Copy_Assignable<T>
{
}

concept Regular<typename T> :
    Semiregular<T>, Default_Constructible<T>, Equality_Comparable<T>
{
}

concept Arithmetic_Like<typename T> :
    Regular<T>, Less_Than_Comparable<T>,
    Has_Unary_Plus<T>, Has_Negate<T>,
    Has_Plus<T>, Has_Minus<T>, Has_Multiply<T>, Has_Divide<T>,
    Has_Preincrement<T>, Has_Postincrement<T>,
    Has_Predecrement<T>, Has_Postdecrement<T>,
    Has_Plus_Assign<T>, Has_Minus_Assign<T>,
    Has_Multiply_Assign<T>, Has_Divide_Assign<T>
{
    explicit T::T(intmax_t x);
    explicit T::T(uintmax_t x);
    explicit T::T(long double x);

    requires
        Convertible<Has_Unary_Plus<T>::result_type, T>
        && Convertible<Has_Negate<T>::result_type, T>
        && Convertible<Has_Plus<T, T>::result_type, T>
        && Convertible<Has_Minus<T, T>::result_type, T>
        && Convertible<Has_Multiply<T, T>::result_type, T>
        && Convertible<Has_Divide<T, T>::result_type, T>
        && Same_Type<
            Has_Preincrement<T, T>::result_type, T&>
        && Same_Type<
            Has_Postincrement<T>::result_type, T>
        && Same_Type<
            Has_Predecrement<T>::result_type, T&>
        && Same_Type<
            Has_Postdecrement<T>::result_type, T>
&\& \text{Same\_Type}<
   \text{Has\_Postdecrement}<T>::\text{result\_type}, T>
&\& \text{Same\_Type}<
   \text{Has\_Plus\_Assign}<T, T \text{ const}\&>::\text{result\_type}, T&>
&\& \text{Same\_Type}<
   \text{Has\_Minus\_Assign}<T, T \text{ const}\&>::\text{result\_type}, T&>
&\& \text{Same\_Type}<
   \text{Has\_Multiply\_Assign}<T, T \text{ const}\&>::\text{result\_type}, T&>
&\& \text{Same\_Type}<
   \text{Has\_Divide\_Assign}<T, T \text{ const}\&>::\text{result\_type}, T&>;
}\

\text{concept Integral\_Like<typename T>}
 : \text{Arithmetic\_Like<T>}
 , \text{Has\_Complement<T>, Has\_Modulus<T>}
 , \text{Has\_Bit\_And<T>, Has\_Bit\_Or<T>, Has\_Bit\_Xor<T>}
 , \text{Has\_Left\_Shift<T>, Has\_Right\_Shift<T>}
 , \text{Has\_Modulus\_Assign<T>}
 , \text{Has\_Bit\_And\_Assign<T>, Has\_Bit\_Or\_Assign<T>, Has\_Bit\_Xor\_Assign<T>}
 , \text{Has\_Left\_Shift\_Assign<T>, Has\_Right\_Shift\_Assign<T>}
{
  \text{requires}
   \text{Convertible<Has\_Complement<T>::result\_type, T>}
&\& \text{Convertible<Has\_Modulus<T, T>::result\_type, T>}
&\& \text{Convertible<Has\_Bit\_And<T, T>::result\_type, T>}
&\& \text{Convertible<Has\_Bit\_Or<T, T>::result\_type, T>}
&\& \text{Convertible<Has\_Bit\_Xor<T, T>::result\_type, T>}
&\& \text{Convertible<Has\_Left\_Shift<T, T>::result\_type, T>}
&\& \text{Convertible<Has\_Right\_Shift<T, T>::result\_type, T>}
&\& \text{Same\_Type}<
   \text{Has\_Modulus\_Assign<T, T \text{ const}\&>::result\_type}, T&>
&\& \text{Same\_Type}<
   \text{Has\_Bit\_And\_Assign<T, T \text{ const}\&>::result\_type}, T&>
&\& \text{Same\_Type}<
   \text{Has\_Bit\_Or\_Assign<T, T \text{ const}\&>::result\_type}, T&>
&\& \text{Same\_Type}<
   \text{Has\_Bit\_Xor\_Assign<T, T \text{ const}\&>::result\_type}, T&>
&\& \text{Same\_Type}<
   \text{Has\_Left\_Shift\_Assign<T, T \text{ const}\&>::result\_type}, T&>
&\& \text{Same\_Type}<
   \text{Has\_Right\_Shift\_Assign<T, T \text{ const}\&>::result\_type}, T&>;
}

\text{concept Signed\_Integral\_Like<typename T>}
 : \text{Integral\_Like<T>}
{
}
A.3 Iterator Concepts

The following concepts are adapted from the C++0x concept proposals for the iterators component of the Standard Library. The concepts defined here are the same (or very similar to) those presented throughout the dissertation.

```cpp
concept Unsigned_Integral_Like<typename T> : Integral_Like<T> {
}

concept Floating_Point_Like<typename T> : Arithmetic_Like<T> {
}

concept Iterator<typename X> : Semiregular<X>, Equality_Comparable<X> {
    Integral_Type difference_type;
    Move_Constructible reference = typename X::reference;

    reference operator*(X&&);
    X& operator++(X&);
    X operator++(X& x, int) {
        X y(x);
        ++x;
        return y;
    }
}

concept Input_Iterator<typename X> : Iterator<X> {
    Object_Type value_type;
    Move_Constructible pointer;

    requires Convertible<pointer, value_type const*>;
    requires Convertible<reference, value_type const&>;
    requires Convertible<
        typename Has_Address_Of<reference>::result_type, pointer>;

    pointer operator->(X&&);
}

concept Output_Iterator<typename X, typename Value> {
    requires Iterator<X>;
    requires */*/
```
requires Has_Assign<typename Iterator<X>::reference, Value>;
}

explicit concept Multipass_Iterator<typename X> {
  requires Iterator<X>;
}

explicit concept Forward_Iterator<typename X> : Input_Iterator<X>, Regular<X>
{
  axiom MultiPass(X a, X b) {
    (a == b) => (++a == ++b) && ((void)++X(a), *a) == *a;
  }
}

concept Bidirectional_Iterator<typename X> : Forward_Iterator<X>
{
  Move_Constructible postdecrement_result;
  X& operator--(X&);
  X operator--(X& x, int) {
    X y(x);
    --x;
    return y;
  }
}

concept Random_Access_Iterator<typename X> : Bidirectional_Iterator<X>, Less_Than_Comparable<X>
{
  X& operator+=(X& x, difference_type n);
  X operator+(X x, difference_type n) {
    x += n;
    return x;
  }
  X operator+(difference_type n, X x) {
    return x + n;
  }
  X& operator-=(X& x, difference_type n);
  X operator-=(X x, difference_type n) {
    x -= n;
    return x;
  }
  difference_type operator-=(X const& x, X const& y);
  reference operator[](X const& x, difference_type n) {
return *(x + n);
}

template<ObjectType T>
concept_map Random_Access_Iterator<T*>
{
typedef T value_type;
typedef pptrdiff_t difference_type;
typedef T& reference;
typedef T* pointer;
}

template<ObjectType T>
concept_map Random_Access_Iterator<T const*> {
typedef T value_type;
typedef pptrdiff_t difference_type;
typedef T const& reference;
typedef T const* pointer;
}

concept Range<typename R> {
    Iterator iterator;
    iterator begin(R&);
    iterator end(R&);
}

template<typename T, size_t N>
concept_map Range<T[N]> {
    typedef T* iterator;

    iterator begin(T (&a)[N]) {
        return a;
    }

    iterator end(T (&a)[N]) {
        return a + N;
    }
}
REFERENCES


Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany, Sep 22-25, pp. 228-244.


Conference on Software Maintenance (ICSM'04), Beijing, China, Sep 28-Oct 4, pp. 57-66.


