Caramel: A Concept Representation System for Generic Programming

Jeremiah Willcock
jewillco@osl.iu.edu

Jeremy Siek
jsiek@osl.iu.edu

Andrew Lumsdaine
lums@osl.iu.edu

Open Systems Laboratory
Computer Science Department
Indiana University
Bloomington, IN 47405
http://www.osl.iu.edu/research/caramel/

ABSTRACT

This paper introduces Caramel, a concept representation markup language system for generic programming in C++. Concepts, as abstract interface descriptions for templates, have a central role in generic programming. However, there is no language support for them in C++ and as a result they are only represented implicitly (in the form of documentation) and have no power to enforce interface adherence. The Caramel system provides a framework for explicit representation of concepts (in XML) and subsequent integration of the concept descriptions into a generic library. Using the XML concept description (as defined by the Caramel DTD), tools in the Caramel system automatically generate concept checking classes (compile time assertions) for interface adherence as well as archetypes (concept exemplars) that can be used for verification of algorithm implementations. Documentation associated with the concept is also generated in a number of popular formats, including HTML and LaTeX. Since the concept checking classes, archetypes, and documentation are all generated from a single source (the XML concept description), consistency between them is assured.

1. INTRODUCTION

Generic programming has continued to evolve as an important paradigm for building robust and reusable software in C++. The term "concept" [4] has emerged to denote specifically the interface description for templates that are at the heart of C++ generic programming frameworks. Although concepts play an obviously critical role in generic programming, they are typically only used implicitly. That is, since there is no language support to describe concepts, they are not represented explicitly in generic libraries. As a result, there is no enforcement that template arguments adhere to the required interface.

More specifically, a *concept* is a set of requirements (valid expressions, associated types, semantic invariants, complexity guarantees) that a type must fulfill to be correctly used within the context of a generic algorithm. A type that fulfills the requirements of a concept is said to *model* the concept. In C++, concepts are represented by formal template parameters to function templates (generic algorithms). However, template parameters are merely placeholders and there is no restriction on the set of valid type arguments. By convention, template parameters are given names corresponding to the concept that is required and the concepts are specified elsewhere in documentation form.

For example, here is the prototype for the `std::sort()` algorithm:

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

For the `std::sort()` algorithm to operate properly, the type of the variables bound to `first` and `last` must meet the requirements of the Random Access Iterator concept, which is specified in Section 24.1.5 of the ISO C++ standard [14].

Unfortunately, although the template system provides the flexibility that makes generic programming possible in C++, the templates themselves carry no particular semantic information relative to a generic algorithm. That is, there is no particular meaning to the compiler conveyed by using the name `RandomAccessIterator` for the template parameter above, nor does the compiler have any means of enforcing — at the point where `std::sort()` is instantiated — that the type bound to `RandomAccessIterator` actually fulfill the requirements of a Random Access Iterator.

Naturally, if a generic algorithm is invoked with a type that does not fulfill at least the syntactic requirements of the concept, a compile-time error will occur. However, this error will not *per se* reflect the fact that the type did not meet all of the requirements of the concept. Rather, the error may occur deep inside the instantiation hierarchy at the point where an expression is not valid for the type, or where a presumed associated type is not available.

What is needed is an explicit representation for concepts and tools that use that representation to enforce interface conformance and to verify algorithm implementations against their interface specifications.

In this paper we describe Caramel, a concept representation markup language system for generic programming in C++. The Caramel system provides a framework for explicit representation of concepts and subsequent integration of the

concept descriptions into a generic library. One of the primary goals for Caramel is to elevate the role of concepts in generic programming, and to thereby provide mechanisms for safer generic programming.

Central to Caramel is a DTD (document type definition) that defines how concepts can be described in XML. Given an XML concept description based on the Caramel DTD, tools in the Caramel system can automatically generate the following three types of output products, each of which has important contributions to better generic programming:

Concept checking classes: These are used as compile time assertions for enforcing interface adherence (e.g., that input types meet the requirements of an algorithm).

Concept archetypes: These are exemplar classes that can be used for verifying algorithm interfaces.

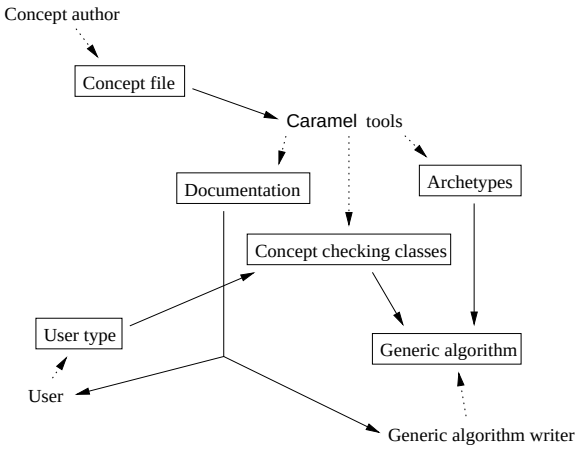
Documentation: These are human readable forms of the concept specification, formatted in familiar ways and given in popular document formats (e.g., HTML).

Moreover, since the concept checking classes, archetypes, and documentation are all generated from a single source (the XML concept description), consistency between them is assured.

A description of the information flow in a Caramel-based generic programming project is shown in Figure 1. In the figure, a *concept author* creates concept descriptions using the Caramel concept format. These are then processed using the tools in the Caramel system to produce concept checking classes, archetypes, and documentation. A *user* creating types to use in generic algorithms can use the documentation to create types modeling the concept, and use the concept checking classes to ensure that the type truly does model the concept. A *generic algorithm author* can use the documentation to create algorithms that use the concepts, the concept checking classes to ensure that only correct types are used in the algorithm, and the archetypes to test that the algorithm uses only properties of a type specified in the concept.

2. CONCEPTS IN GENERIC PROGRAMMING

A concept is used to represent the interface between a generic algorithm and the types that may be provided to it as template parameters. If a generic algorithm requires its parameters to model specific concepts, and the data types given to it as parameters model those concepts, the algorithm will compile and run correctly (provided that the algorithm itself is implemented correctly). A concept contains both syntactic and semantic requirements. An example of syntactic requirements is that a generic binary search algorithm must be able to determine the length of the sequence to be searched (using a function or method), get a specific element of it, and compare two elements. An example of semantic requirements is that the sequence must be sorted, and that the comparison operator used for the sort be the same as that used to compare elements in the binary search.



Dotted lines indicate production, solid lines indicate use.

Figure 1: Information flow in Caramel

A concept thus consists of constraints, which are individual requirements on functions, operators, and runtime behavior. The constraints require certain properties of the arguments of the concept, including both compile-time and run-time properties.

Several kinds of constraints are found in concepts. The first kind expresses some operation (or method, or a lookup table entry) that must be valid on the type(s) in the concept. These can require that operations be defined on either types or values. In addition, *associated types* can be defined. These require that some type exist that is related to another. Associated types are often found in traits classes [24], but do not need to be located there. An associated type definition also provides a shorthand name for the type, so it can be referrred to more easily later in the concept description.

In C++ concepts, constraints requiring operations to be present are usually called *valid expressions*. A valid expression consists of an expression that must compile using the given types, and whose return type must have certain properties. For instance, a valid expression might require that two values of a the type `T` must be addable using the `+` operator, and that the result of this addition must also be of type `T`. A valid expression may also specify pre- and postconditions, as well as the semantics of the expression.

Constraints in a concept can also take the form of *invariants* or identities. This is a requirement that some expression always be true for all values of a type. For instance, a parallel prefix sum algorithm will require that the operation used to combine the elements of a list is associative. A concept may also provide guarantees about the resource complexity of various operations.

Concepts may also include *refinements*. A refinement includes the requirements of another concept into the current one. For instance, the Bidirectional Iterator concept refines the Forward Iterator concept. This means that every type that models Bidirectional Iterator must be a model of Forward Iterator. A refinement provides a subset relationship

between the set of types modeling the refining concept and the set of types modeling the refined concept.

In order to be automatically processed, concepts must be represented in a computer-readable format. This format should be easy to parse, and contain all of the information for documentation. It should also have an description of the valid expressions and other C++ code in a way that simplifies code generation for concept checking classes and archetypes.

A concept representation must contain, first and foremost, the constraints that form the "meat" of the concept. Human-readable documentation must also be provided. This includes the overall documentation of the concept as well as definitions, algorithm complexities, and other pieces of text. Notations (variable definitions) are used to simplify the output of concept documentation and concept checking classes.

2.1 Related Work

2.1.1 Interfaces

Several languages provide support for *interfaces* (in the sense of a Java interface). An interface (also called an abstract base class) specifies the methods required for a parameter to a function or method. A user-defined type then "implements" (or inherits from) the interface. This specifies to the compiler that the user type may be used where parameters implementing that interface are required, and causes the compiler to check the user type for the methods required by the interface. If the user type does not have all of the methods required by the interface, a compile-time error occurs. The advantage of this method is that the compiler tests the user type for methods, and does not require a separate concept checking library. The disadvantages are that only methods can be checked, and that the user must explicitly state which interfaces will be implemented in the user type. Interfaces can be used both to constrain subtype polymorphism (ensure that an object passed to a method has the correct virtual functions implemented) and to constrain parametric polymorphism (ensure that a type passed to a generic algorithm has the correct methods defined). Interfaces exist in the C++ [14] and Java [12] languages, among others, to constrain subtype polymorphism, and are used to constrain generic type parameters in the Generic Java [8] and Pizza [27] languages. The Unified Modeling Language [25], a format for specifying the designs of object-oriented programs, also supports interfaces. Designs written using it can also be represented in XML. The C++ Standard Template Library is modeled with UML in [10].

2.1.2 Interface Definition Languages

Interface definition languages are used to specify the interfaces to objects accessed using a system such as CORBA [26] or COM [21]. The interface definition language often contains more information about an interface than what a programming language specification contains, since a remote object protocol needs to know which parameters are read, which are written, and other data access requirements. Therefore, an interface definition language specifies the interface to a type. Both the CORBA IDL [1] and Microsoft IDL [22] (used in COM and DCOM) contain interface specification systems.

2.1.3 Signatures

Similar to an interface, a *signature* (also called a type class) is a list of methods that a type must support. Unlike an interface, however, a user type does not need to explicitly state which signatures are supported by a type. The compiler checks that the user type supports the correct methods when the type is passed to a function. Signatures are supported in the GNU C++ compiler g++ [5] and in the Theta [9] programming language. In the functional programming community, the Haskell [15] and Standard ML [23] languages use signatures to constrain generic type parameters. In addition to the use of interfaces mentioned above, the UML also supports "types," which are equivalent to signatures [25].

2.1.4 Assertions and Design-by-Contract

Some programming languages provide methods to verify at runtime that a type satisfies certain properties. This is usually done by inserting extra debugging code, called an *assertion*, that tests the properties and causes the program to abort if they are not satisfied. These can be used, for example, to document the semantics of every method on a data structure and ensure that a particular implementation follows the specification. This is called "design-by-contract" [20]. In structure, these are very similar to interfaces in that they specify the presence of a list of methods. However, they also provide detailed semantics of the methods. Eiffel uses these for both non-generic and generic classes, including restricting parameters of generic types to fit a particular interface.

Other systems, such as the Anna preprocessor for Ada [19], provide a more general specification language that can be embedded in a program. A preprocessor then uses this to generate documentation, assertions within the code, or input to a formal correctness proof system. These provide a way to specify the interfaces to a generic algorithm and generate code and documentation from them.

2.1.5 Formal Specification

Another way to specify the interface to a type is a formal specification. In particular, several formal specification languages support generic programming. These languages include fully formal descriptions of type constraints, in much more detail than Caramel provides. However, they are not designed to have their specifications converted into documentation or programs in standard programming languages. Examples of formal specification languages that are generic include Tecton [16] and OBJ3 [11]. These languages allow a user to prove mathematically that a certain type matches the concept, and thus that a generic algorithm will function correctly.

3. CONCEPT APPLICATIONS

3.1 Concept Checking Classes

Because C++ compilers do not support concepts, we previously developed a C++ idiom (which we call concept checking [29]) for defining compile-time assertions that a type model a concept. A library supporting this idiom, and which provides concept checks for the Standard Library, is available as the Boost Concept Checking Library (BCCL) [7].

The following is a concept checking class for Less Than Comparable. The required valid expressions for the concept are exercised in the `constraints()` member function.

```
namespace boost {
  template <class T>
  struct LessThanComparableConcept {
    void constraints() {
      (bool)(a < b);
    }
    T a, b;
  };
}
```

This concept checking class is instantiated with the user's template arguments at the beginning of the generic algorithm using BCCL `function_requires()`. An example of this is given for a safe version of the STL `sort` function.

```
#include <boost/concept_check.hpp>
template <class Iter>
void safe_sort(Iter first, Iter last)
{
  typedef typename std::iterator_traits<Iter>
    ::value_type T;
  boost::function_requires<
    boost::LessThanComparableConcept<T> >();
  // other requirements ...
  std::sort(first, last);
}
```

In the Caramel system, a C++ program converts a concept in XML into a concept checking class, which is described in Section 4.2.1.

3.2 Concept Archetypes

The complementary problem to concept checking is verifying that the documented requirements for a generic algorithm actually match the algorithm's implementation — a problem we refer to as *concept covering*. Typically, library implementors check for coverage by manual inspection, which is error prone. We previously developed a C++ idiom that exploits the C++ compiler's type checker [29], which is also available as part of the Boost Concept Checking Library. The BCCL provides an *archetype class* for each concept used in the Standard Library. An archetype class provides a minimal implementation of a concept. To check whether a concept covers an algorithm, the archetype class for the concept is instantiated and passed to the algorithm.

For example, suppose that the documentation for `std::sort()` stated that the iterator arguments' type must model Random Access Iterator, and that the value type of the iterator must be Less Than Comparable. That these concepts do indeed cover `std::sort()` can be verified by compiling the following program.

```
#include <algorithm>
#include <boost/concept_archetype.hpp>
int main() {
  typedef
    boost::less_than_comparable_archetype<> T;
  boost::random_access_iterator_archetype<T> ri;
  std::sort(ri, ri);
}
```

Compiling this program will result in a compiler error because as it turns out, those concepts do not cover `std::sort()`. The resulting error message indicates that the algorithm also requires the value type to be Copy Constructible. Not only is the copy constructor needed, but the assignment operator is needed as well. This could either be an error in the implementation of `sort()` (it uses an operation that is not truly necessary) or an error in the interface (the requirements for `sort()` do not provide enough features to correctly implement the algorithm). In this case, the problem is with the interface specification, but this must be analyzed and evaluated on a case-by-case basis. These requirements (copy construction and assignment) are summarized in the Assignable concept. The following code shows the implementation of the archetype class for Assignable. The `Base` template parameter is provided so that archetypes can be combined. To check `std::sort()`, the archetype classes for Assignable and Less Than Comparable would need to be combined.

```
namespace boost {
  template <class Base = null_archetype<> >
  class assignable_archetype : public Base {
    typedef assignable_archetype self;
  public:
    assignable_archetype(const self&) { }
    self& operator=(const self&)
      { return *this; }
  };
}
```

The Caramel system generates archetype classes from the XML concept descriptions, as described in Section 4.2.2.

3.3 Documentation Generation

The third product of the concept representation system is documentation for each concept. The purpose of concept documentation is to communicate interface requirements to the user. The Caramel system's concept documentation is modeled after the concept descriptions in Austern's *Generic Programming and the STL* [4] and the SGI STL documentation [31]. Documentation is produced in both HTML and LaTeX formats.

The documentation for each concept contains several parts: header information, various sections of human-readable documentation, notation used in the concept, a list of associated types, a list of valid expressions, a list of invariants, and a see-also section with a list of related concepts. Other information, such as footnotes and copyright messages, are also inserted where appropriate.

Several features are required of the concept description for documentation generation. All information must be independent of the output documentation language, so it is easy to add additional languages. In the Caramel system, there is no output-specific code in the XML concept descriptions. For example, mathematical expressions are represented using a subset of MathML [3] which is later converted into each specific output language. Another requirement on the concept description is that the structure must be somewhat similar to the documentation. For instance, the valid expression lists are modeled closely after what the documentation should look like, so the translation programs do not need to be overly complicated.

The process by which the Caramel system converts XML concept descriptions into documentation is described in Section 4.2.3.

There are several types of related work in automated documentation generation. Systems such as Javadoc [32] and SWIG [6] generate documentation from source code (with special comments and other markup). These are not designed to document concepts, but Javadoc has been used to document the interfaces in the Java system. A documentation tool that is designed for generic programming and concepts is HDoc [13]. It is intended for documenting Haskell programs, and can automatically generate documentation for a type class (equivalent to a concept) from marked-up source code. Also, a system such as Rational SoDA [28] can generate documentation from source code and keep it updated as the source code changes. A logical extension of this is literate programming, in which the program is interspersed within large blocks of documentation. One of the first literate programming systems was Knuth's WEB system [18].

4. THE CARAMEL SYSTEM

The Concept Representation Markup Language (Caramel) system uses XML [35] to represent concepts. The format is formally defined in an XML document type definition (DTD). Tools are provided to convert from this format to HTML and LaTeX documentation, as well as to produce automated concept checking classes and archetypes. An overall diagram of the steps in each process is in Figure 2.

4.1 The Caramel Concept Format

The Caramel concept format contains all of the elements listed in section 2 as part of a generic concept description. Other elements are also added to simplify the generation of documentation, concept checking classes, and archetypes.

4.1.1 Sections of a Concept

A concept described in the Caramel format consists of many sections. A list of the sections is given first, followed by detailed descriptions.

1. Concept name
2. Parameters
3. Copyright statements
4. Models sentence
5. Documentation
6. Notation
7. Refinements
8. Associated types
9. Defined types
10. Valid expressions
11. See-alsos
12. Example models

4.1.1.1 Concept Name

Each concept must have a name and optionally a category. The name has each constituent word capitalized. A separate name is used in actual code, and must conform to C++ naming standards. This name is generated by removing white space and hyphens from the human-readable name. For instance, in the Hilbert Space concept, the name is Hilbert Space while the name used in code is `HilbertSpace`.

4.1.1.2 Parameters

Most often a concept describes the behavior of a single type, but sometimes a concept describes interactions between multiple types. For instance, the Input Iterator concept has only one parameter (the iterator type) while the Group concept has four (the element type, the group operation type, the identity operation, and the inverse operation). We refer to these types as the parameters of the concept. Each parameter has a name (which is used to refer to it in the XML description and in C++ code) and a role (which is used to describe the parameter in the documentation). For example, here is a table of the Group concept's parameters:

Name	Role
`T`	element-type
`Op`	operation-type
`IdentityOp`	identity-function-type
`InverseOp`	inverse-function-type

4.1.1.3 Copyright Statements

One or more copyright statements may be included in each concept, and may be specified either directly in the concept file or as a reference to an external XML file. The copyright information is printed in all files output from the concept.

4.1.1.4 Models Sentence

A models sentence is a template for the sentence used in the documentation to describe a set of types that model a specific concept. This is used to improve the readability and quality of documentation. As an example, the Group concept has a models sentence of "Element type `<arg num="1"/>`, operation `<arg num="2"/>`, identity function `<arg num="3"/>`, and inverse operation `<arg num="4"/>` together must form a `<self/>`." The `arg` tag in this template is replaced with the actual type used as the corresponding parameter of the concept. The `self` tag is replaced with the name of the current concept. This sentence will be substituted with the names of specific types when this concept is referred to.

4.1.1.5 Documentation

Several sections in the concept format express various pieces of human-readable documentation. The main one is named "documentation" which is for providing the main description of the concept and its purpose. There are also "definition", "complexity", and "invariant" sections. These are written with a minimum of formatting, but can contain code blocks and mathematical notation written with an extended subset of MathML [3].

4.1.1.6 Notation

The notation section defines type names and variable names for the concept documentation and concept checking class to use. The most common form is to declare a list of variables of a given type. These variables will be listed in the documentation, and valid expressions needing variables of this type will sequentially use names from the declaration list. For instance, after declaring variables `x` and `y` of type `int`, an expression requiring the sum of two `int`'s will be output as `x + y`.

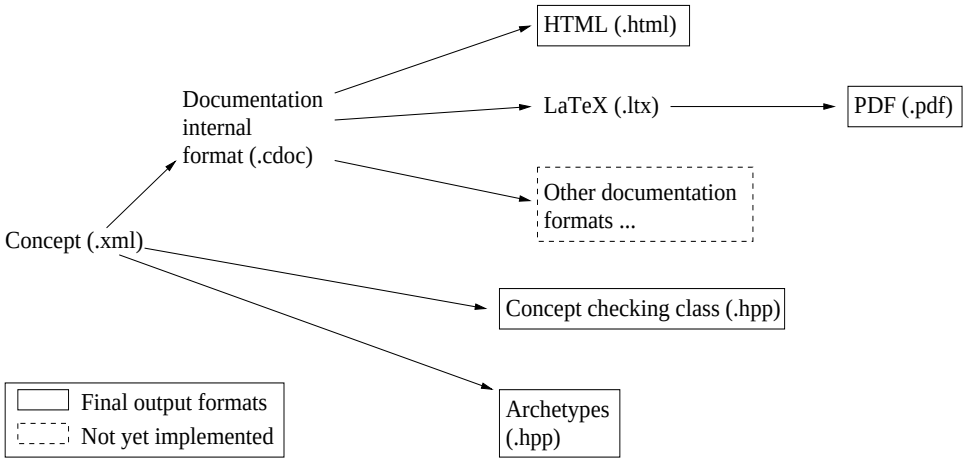


Figure 2: Steps in the Caramel processing structure

4.1.1.7 Refinements

A concept may also refine another concept. For instance, if a concept `B` refines a concept `A`, all of the requirements specified by `A` will also appear in `B`. This is reminiscent of a subtype relationship between the concepts. A normal refinement requires that all parameters be exactly the same between the concepts. This is inadequate for some concepts, so an alternative form allows the use of a subset of the parameters. This form uses the models sentences described earlier.

4.1.1.8 Associated Types

An associated type is a type that is not directly a parameter of a concept, but is used as part of the concept's definition. For example, an STL iterator must have an associated type for its value type (the type of data pointed to by it). In the case of an iterator, the associated types are provided using a traits class, but other concepts have associated types that are member types of a concept parameter directly. The associated `value_type` of an iterator can be accessed through the `std::iterator_traits` class. Each description for an associated type includes its name and the required access method (nested typedef or particular traits class).

4.1.1.9 Defined Types

Defined types are a notational convenience for the XML concept descriptions. They create a short name for a type which can then be used in other parts of the XML concept description. Defined types do not appear in the output, but instead behave more like macros.

4.1.1.10 Valid Expressions

One of the most important parts of a concept specification is the list of valid expressions. These come in two types (listed separately in documentation): value expressions and type expressions. They have similar semantics, with the main difference being whether the constraint is on a type or a value. Each of these constraints consists of a C++ expression (written as a parse tree) and a list of requirements on the return type of the expression. These requirements can include same-type constraints (this type must be the same as some other specified type), convertible-to constraints, assignable-to constraints, and derived-from constraints. In addition, return types of expressions may be required to model some other concept (possibly with other types specified as other arguments for the concept). These (as well as invariants) form most of the actual requirements on the types in a concept.

4.1.1.11 See-alsos

Another feature a concept may have is a list of other concepts related to it. This is purely for documentation purposes.

4.1.1.12 Models

This section lists examples of types (or sets of types) that satisfy the concept. This section is purely for documentation purposes.

4.1.2 Concept XML Definition

The concept structure described above is defined using an XML document type definition (DTD). This allows the use of tools such as Xerces [2] to validate whether a particular concept description has the correct format. In order to allow more flexibility in the section ordering, the DTD does not express all constraints in the concept format, but can catch most major errors.

4.1.3 Parse Trees

One interesting feature of the Caramel format is that C++ code in valid expressions is expressed as a parse tree. This simplifies the production of archetypes (which use parse trees and do not directly translate them to C++ code) at the cost of requiring a parse-tree-to-code converter for producing documentation and concept checking classes. However, this is made up for by the expandability provided by this converter. For instance, the converter can be set to ignore top-level `const` and `volatile` qualifiers, and to ignore top-level references. This parse tree format is somewhat like

that used in GCC-XML [17], but covers the details of C++ expressions rather than overall program structure. As an example, the parse tree (expressed in XML) given in Figure 3 represents the extraction of the value type from an iterator named `Iter`.

```
<get-member-type name="value_type">
  <apply-template name="std::iterator_traits">
    <type name="Iter"/>
  </apply-template>
</get-member-type>
```

Figure 3: Parse tree for iterator value types

4.1.4 Concept Mutability

Each concept also has mutable and non-mutable forms. For example, a non-mutable iterator to a container may be moved within the container, but may not change the element it points to. A mutable iterator, on the other hand, is allowed to change the element it points to. These two forms are handled within one concept, with flags at compile-time to determine which form to use. Concept checking classes and archetypes are generated for both forms separately, but documentation has only one version for both concept forms.

4.2 Modules of the System

In addition to the concept structure and DTD described above, the Caramel system contains several programs that operate on the concepts to produce various types of output. The outputs include HTML and LaTeX documentation (with an index in HTML), concept checking classes, and archetypes. All modules are written in C++, and use the GNOME XML library for XML parsing and generation services [34].

4.2.1 Concept Checking Class Generation

To generate automatic concept checking classes, a one-stage process is used. This stage shares several modules with the documentation generator. All sections of documentation intended for human reading are deleted, and each constraint (such as a valid expression) is converted into code to test it. These tests are made much simpler by the use of the Boost concept checking, type traits, and static assertion libraries [7]. The converter produces two concept checking classes (one for the mutable form of the concept, and one for the non-mutable form) intended to be used with the Boost concept checking library.

4.2.2 Concept Archetype Generation

The process used to generate archetypes is much more complicated than the previous one. A roughly three-stage process is used, with the stages integrated into a single C++ program. The three stages are mostly for simplicity, and not an inherent part of the archetype generation process. All operators and functions required by the concept are produced, with dummy values used as results. Default constructors, copy constructors, and assignment operators not explicitly required by the concept are marked "private" to ensure that nothing depends on them implicitly.

4.2.3 Documentation Generation

The process of documentation generation is broken into two steps to ease the addition of new output formats. First, the concept is converted into a language-independent intermediate format, and then into the output format (either HTML or LaTeX in the current implementation).

The first step follows the list of concept sections given above, and gathers information for each section. Documentation is copied to the intermediate format as-is, and other parts (such as C++ code and valid expression return type constraints) are converted to text from their original tree representations. This format matches the structure of the output documentation very closely, and is designed to be easy to parse by an XSLT stylesheet[1] (even though currently it is parsed by a C++ program). Tree conversions are handled by a separate module included by the documentation translation programs.

After this, an output conversion program is run. The current implementation contains two programs for this stage, one for HTML and one for LaTeX output. However, a DTD is provided for the intermediate format so users can write their own processors to handle this stage of the conversion. The programs generally convert tags in the intermediate file directly into output, and do not perform any complicated processing of the data.

4.3 Example

As an example, the Equality Comparable concept from the SGI STL documentation [31] will be traced through the Caramel system. First, an XML file is created for the concept, as shown in Figure 4.[2] Note that this concept description does not include (or require) all of the available concept elements. A concept is described within the scope of a `concept` tag which contains the concept name and the category of the concept (in this case "Utility" for STL utility concepts). There is then one parameter, the type that must be comparable for equality. Following some documentation, two notations `x` and `y` are declared as values (denoted by `sample-value`) of type `X`. Next are two valid expressions. Each requires a specific operator (`==` or `!=`) on values of type `X` whose return type is convertible to `bool`. These are written out in words (`equal-to` and `not-equal-to`) in the parse tree. Finally are some examples (from the SGI documentation) of types modeling the Equality Comparable concept.

From this, a concept checking class is produced, as shown in Figure 5. Note that this section of code follows the concept description fairly closely. The name of the concept is used as the name of the checking structure. The notations in the concept become variable declarations and each valid expression becomes a test in the concept checking class. The first test, for instance, tests whether an equality operator exists for type `X`, and whether the result of this operator is convertible to `bool`. Thus, the concept checking class is a literal translation of the concept description and captures exactly what the concept requires. The second test is similar, but

[1] A stylesheet is the XSLT equivalent to a program. It consists of a set of *templates* (procedures) to handle various types of input.

[2] For the sake of brevity and clarity, this and other examples are shown in an abbreviated fashion.

```
<concept name="Equality Comparable" category="Utility">
  <param name="X" role="comparable-type"/>
  <!-- Copyright statements, other misc. documentation -->

  <models-sentence>The type <arg num="1"/> must be a model of
  <self/>.</models-sentence>

  <notation variables="x y"><sample-value><type name="X"/></></>

  <valid-expression name="Equality test">
    <equal-to>
      <sample-value><type name="X"/></>
      <sample-value><type name="X"/></>
    </equal-to>
    <return-type><convertible-to><type name="bool"/></></>
  </valid-expression>

  <valid-expression name="Inequality test">
    <not-equal-to>
      <sample-value><type name="X"/></>
      <sample-value><type name="X"/></>
    </not-equal-to>
    <return-type><convertible-to><type name="bool"/></></>
  </valid-expression>
  <!-- Example models -->
</concept>
```

Figure 4: Equality Comparable concept file

```
// Copyright statements, header inclusions, namespace declarations
template <class X>
struct EqualityComparableConcept {
  X x, y;
  void constraints() {
    // Equality test:
    (bool)(x == y);
    // Inequality test:
    (bool)(x != y);
  }
};
// Identical struct for mutable form of concept
```

Figure 5: Equality Comparable concept checking class

tests for an inequality operator. In concept checking classes, the mutable and non-mutable forms of a concept are placed in the same file, but only the non-mutable form was shown in this example.

In addition, an archetype is produced, as shown in Figure 6. The figure shows only the non-mutable form, but the mutable form of this concept is the same. Here, a group of types is defined in a separate namespace. The most important is `nt_X`, which corresponds to the type `X` in the concept description. This type, however, does not have any public constructors defined. The concept did not require any, so the archetype must not have them (so that nothing depends on them implicitly). The two helper types, `t1` and `t5`, are convertible to `bool` (but cannot do anything else). The `==` and `!=` operators are defined on objects of type `nt_X` (using references in order to not require a copy constructor), and which return one of the helper types (also by reference). They cannot return `bool` directly, since the concept does not require that. The `sample_value` function used in the archetype just returns a reference to a (nonsense) value of the given type without invoking any constructors. After this, the parameters to the concept are listed in order to test the archetype more easily. Note that the archetype only needs to compile successfully, so its runtime behavior is unimportant.

```
// Header and copyright information
namespace boost {
  namespace archetypes {
    // Forward declarations of structs
    // Actual archetype class definition
    struct EqualityComparable_mutable {
      // Typedefs for structs
      typedef EqualityComparable_mutable_realstructs::nt_X nt_X;
      typedef EqualityComparable_mutable_realstructs::t1 t1;
      typedef EqualityComparable_mutable_realstructs::t5 t5;
      typedef nt_X parameter_1; // Concept parameter
    };

    // Structure definitions
    namespace EqualityComparable_mutable_realstructs {
      struct nt_X // No operations (all explicitly disabled).
      struct t1 // Convertible to bool (other operations disabled).
      struct t5 // Convertible to bool (other operations disabled).
    }

    // Standalone functions and operators
    namespace EqualityComparable_mutable_realstructs {
      t1& operator==(const nt_X&, const nt_X&) {
        return boost::sample_value< t1 >(); }
      t5& operator!=(const nt_X&, const nt_X&) {
        return boost::sample_value< t5 >(); }
    }
  }
}
```

Figure 6: Equality Comparable archetype

The last major type of output that the Caramel system can generate is concept documentation, as shown in Figure 7. Like the concept checking class, this follows rather directly from the concept XML document. First, there is a heading with the concept name and category followed by the description. After this, a notation section with both the concept parameters and the notation from the concept is shown. Finally are lists of valid expressions, example models (in this case, from the SGI documentation), and copyright messages.

5. LIBRARIES AND FRAMEWORKS DOCUMENTED USING CARAMEL

As part of the development of the Caramel system, we developed XML concept descriptions for several libraries. This was beneficial in expanding the range of concepts that the Caramel system was able to represent.

The C++ Standard Library

Almost all of the concepts of the C++ standard library were documented using the Caramel system. Based on the SGI STL documentation [31] and the C++ language standard [14], the iterator, function object, container, and utility concepts were documented. The Equality Comparable concept given earlier as an example is from this library. These concepts exposed several weaknesses in early versions of the Caramel system. An example of one was dealing with member access through pointers using `operator->`. The way in which this operation is implemented is unique, since only part of the operation is provided by the type (returning a pointer) and the rest (accessing the member) is handled by the C++ implementation. The issue was solved by handling `operator->` differently from the other operators in the conversion from the XML concept description (`.xml`) to the intermediate documentation format (`.cdoc`). The operator is defined in concepts as returning a pointer, which is shown in concept documentation with an explanatory note.

Equality Comparable
Category: Utility

Description:
EqualityComparable types must have `==` and `!=` operators.

Notation:

`X`	A type playing the role of comparable-type in the Equality Comparable concept.
`x, y`	Objects of type `X`

Valid expressions:

Equality test

Expression:	`x == y`
Return type:	Convertible to `bool`

Inequality test

Expression:	`x != y`
Return type:	Convertible to `bool`

Models:
- `int`
- `std::vector<int>`

Figure 7: Equality Comparable documentation

The Boost Property Map Concepts

The four Boost property map concepts were documented using the Caramel system. A property map is an association between a key set and a value set. There is no requirement that the key set be iterable (or finite). Property maps can be read-only, write-only, read-write, or provide references to the associated value. Property maps are generalizations of arrays or STL maps. These did not require any substantial changes to the system, but were done fairly early in the development cycle, so they influenced what features were included in it. The concepts in this category are Readable Property Map, Writable Property Map, Read/Write Property Map, and Lvalue Property Map. We plan on applying this system to the rest of the Boost libraries in the future.

The MTL Algebra Concepts

All of the algebra concepts from a draft of the Matrix Template Library [30] were documented using the Caramel system. These concepts demonstrate different features of the system, since they include many mathematical formulas and often have concepts taking multiple type parameters. The Hilbert Space concept used as an example is from this library. Mathematics, especially the field of algebra, uses ideas very similar to concepts. The analog to concepts is algebraic structures, which have defined properties and about which theorems are proved. A full description of the algebraic structures is given in [33], among other sources. The concept descriptions from MTL are basically translations of the algebraic structures into C++ code. These concepts caused several problems for the Caramel system. The concepts use large amounts of mathematical notation, which lead to the use of MathML [3] to represent it. Because mathematical concepts often have many types as parameters, the "models sentence" construct described earlier needed to be added to make these concepts' documentation more readable. In addition, concepts in this group refer to themselves, and so loops needed to be detected when expanding archetype constraints to avoid infinite recursion. Also, some problems in the concepts themselves were found. For instance, the MTL definition for the Additive Abelian Group concept required that the result of an addition be convertible to the original type, but did not require that of the result of *two* additions. This was changed in the Caramel description of the concept. Future versions of the MTL are likely to include concepts documented using the Caramel system. The following concepts from this library were documented:

Additive Abelian Group	Finite Vector Space
Banach Space	Hilbert Space
Commutative Groupoid	Linear Algebra
Commutative Ring	Linear Operator
Field	R-Module
Finite Banach Space	Ring
Finite Hilbert Space	Transposable Linear Operator
Finite Linear Operator	Vector Space

6. TESTING PROCEDURES

As a test of the concept checking classes and archetypes produced by the Caramel system (and thus also of the concept descriptions underlying them), we also tested each STL container class against the appropriate concept checking classes and tested each STL algorithm with an archetype of the concept required of that algorithm's arguments. All of the algorithms passed with their corresponding archetypes, and almost all of the containers passed (the one that failed was due to a bug in the version of the standard library used for the tests).

7. FUTURE WORK

One straightforward avenue for improvement to the Caramel system will be to document more concepts using it. This will provide valuable feedback and improve the appeal of the system to potential users.

One possible new feature is to add support for some constructs that the current Caramel system does not support. The system currently does not support member class templates or non-type template parameters, and adding them would allow a more general range of concept to be expressed. However, they are not needed for any concepts currently using the system. Member class templates are required to document the C++ Standard Library allocators [14] (memory allocation routines). These features would probably not be too difficult to implement if they were ever needed.

Another possible new feature is to increase the formality of the concept specifications in the Caramel system. For instance, a formal proof language such as Tecton [16] could be one of the output formats, and thus could prove that a particular class models a certain concept. This would require substantial changes to the system, as currently many parts of a concept (such as invariants and complexity specifications) are written for documentation only and are not interpreted by Caramel. In addition, all of the concepts would need to be rewritten to use the new formal definitions. There are also probably some concepts whose current documentation lacks certain invariants that are implied (or can be reasonably assumed by a human reader) that would

need to be explicitly stated. This would be a major change to the Caramel system.

A feature to ease concept entry would be a terser, more readable input language that is converted by a program into XML. This would be similar to the language used by Javadoc [32] or HDoc [13], but specialized for C++ concepts. Currently, entering a concept requires much typing (or copy-and-paste from other concepts) because XML is a verbose language. The two most verbose sections of a concept currently are C++ parse trees and mathematical notation. Parse trees use one tag to represent each operator, and so require the name to be specified at both the start and end of each node. Mathematical notation, expressed in an extended subset of content MathML [3], is also written as a parse tree and is thus similarly verbose. A better system would use a more readable format, and have a preprocessor to convert this into XML for processing. XML would still be used as the internal format so that programs would have an easier time parsing the concepts.

An alternative way to simplify concept entry would be to have a specialized concept editor. This editor would provide a template for the concept, and would allow expression-based entry for C++ parse trees and mathematics (according to the MathML Web pages [3], such editors already exist for that language). A concept editor would likely be WYSIWYG, and show the concept's documentation on the screen. However, what would be written into the file would be a full XML description of the concept. It is possible that existing XML editors could be extended for this purpose.

The support of other generic programming languages would also be a useful feature for the Caramel system. Currently only C++ is supported, but the kinds of information contained in a concept description are not very dependent on the particular language. The set of operators supported is, but most other concept features are not. Therefore, in the future, the Caramel system could be extended to generate signatures, type classes, and documentation for multiple generic programming languages.

A release of the Caramel system is expected in the future. It will be available at `http://www.osl.iu.edu/research/-caramel/`.

8. ACKNOWLEDGMENTS

The authors would like to thank the following individuals for many helpful discussions that contributed to Caramel: Matt Austern, Alex Stepanov, David Musser, Sibylle Schupp, and Ron Garcia. This work was supported by NSF grant ACI-9982205. The first author was supported by a Schmitt fellowship at the University of Notre Dame during the course of this work. This paper benefitted from the comments of several anonymous reviewers.

9. REFERENCES

[1] Information technology – open distributed processing – interface definition language. ISO/IEC 14750:1999.

[2] Apache XML Project. *Xerces-Java*. `http://xml.apache.org/xerces-j/index.html`.

[3] R. Ausbrooks, S. Buswell, S. Dalmas, et al. *Mathematical Markup Language (MathML) Version 2.0*. World Wide Web Consortium, February 2001. `http://www.w3.org/TR/MathML2`.

[4] M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[5] G. Baumgartner and V. F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice and Experience*, 25(8):863–889, August 1995.

[6] D. Beazley et al. *Simplified Wrapper and Interface Generator (SWIG)*. `http://www.swig.org/`.

[7] Boost. *Boost C++ Libraries*. `http://www.boost.org/`.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, 1998.

[9] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA*, pages 156–158, 1995.

[10] H. Eichelberger and J. W. v. Gudenberg. UML description of the STL. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[11] J. A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, Sept. 1984.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[13] A. Groesslinger. *HDoc*, February 2001. `http://www.fmi.uni-passau.de/~groessli/hdoc/`.

[14] International Standardization Organization (ISO). *ANSI/ISO Standard 14882, Programming Language C++*. 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland, 1998.

[15] S. P. Jones, J. Hughes, et al. *Haskell 98: A Non-strict, Purely Functional Language*, February 1999. `http://www.haskell.org/onlinereport/`.

[16] D. Kapur, D. R. Musser, and A. A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Proceedings of a Workshop on Program Specification*, volume 134 of *LNCS*, pages 402–414, Aarhus, Denmark, Aug. 1981. Springer.

[17] B. King. *GCC-XML: An XML output extension to GCC's C++ front-end*. Kitware, Inc., February 2001. `http://public.kitware.com/GCC_XML/`.

[18] D. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford, CA, 1992.

[19] D. C. Luckham and F. W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[20] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1997.

[21] Microsoft Corporation. *COM*, December 2000. http://msdn.microsoft.com/library/psdk/com/-comportal_3qn9.htm.

[22] Microsoft Corporation. *Microsoft Interface Definition Language*, December 2000. http://msdn.microsoft.com/library/psdk/midl/-ov-iface_9x2r.htm.

[23] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[24] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[25] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, March 2000. http://www.omg.org/technology/documents/-formal/unified_modeling_language.htm.

[26] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, February 2001.

[27] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[28] Rational Software Comporation. *Rational SoDA*. http://www.rational.com/products/soda/.

[29] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[30] J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[31] Silicon Graphics, Inc. SGI implementation of the Standard Template Library. http://www.sgi.com/tech/stl/.

[32] Sun Microsystems, Inc. *Javadoc 1.3*. http://java.sun.com/j2se/1.3/docs/tooldocs/-javadoc/index.html.

[33] B. L. van der Waerden. *Algebra*. Frederick Ungar Publishing, 1970.

[34] D. Veillard. *The XML C Library for GNOME*, June 2001. http://www.xmlsoft.org/.

[35] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000. http://www.w3.org/TR/REC-xml.