Facilitating Software Maintenance by Automated Detection of Constraint Violations

Anir Chowdhury and Scott Meyers

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-37
September 1993

Facilitating Software Maintenance by Automated Detection of Constraint Violations

Anir Chowdhury*
anc@cs.brown.edu

Scott Meyers
sdm@cs.brown.edu

Department of Computer Science
Brown University
Box 1910
Providence, RI 02912

Abstract

In this paper, we describe CCEL, a language that allows programmers to formally express constraints on their software systems and to automatically detect violations of these constraints. We demonstrate the power, the flexibility, and the overall utility of CCEL by showing how it can express real constraints from real software developers for real systems.

1 Introduction

Software teams are invariably bound by a set of constraints when designing and coding complex software systems. Constraints are imposed on a system's design and implementation to ensure that the product is reliable, readable, maintainable and portable. Unfortunately, because most constraints are typically difficult or impossible to formalize, people use their minds as the primary storage for these constraints and write down only a small subset of them.

Even if constraints are written down, there is usually no way to automatically enforce design and implementation standards. Effective maintenance of the resulting systems — ones that fail to follow a consistent, formally expressed, automatically enforced scheme — is problematic at best, impossible at worst. The difficulties often increase when maintenance personnel add to existing code for revision or extension purposes. There is therefore a practical need to formally express the constraints that software teams impose on their designs and implementations.

In this paper, we describe how a wide variety of programmer-defined constraints can be both formally expressed and automatically enforced for software systems written in the C++ language. The fundamental ideas behind our approach, however, are not specific to C++, and the development of similar constraint systems for other programming languages would not be difficult.

* Author's current affiliation: Open Environment Corporation, Cambridge, MA 02139 (anir@oec.com)

Consider the following sample constraints:

- A **stylistic constraint**: *All class names must begin with an upper case letter.* Most software development teams adopt some type of naming convention for identifiers, violations of which range from irritating to misleading.
- An **implementation constraint**: *If a class declares a pointer data member, it must also declare an assignment operator and a copy constructor.* Failure to adhere to this constraint frequently leads to incorrect program behavior [3].
- A **design constraint**: *The member function M in class C must be overridden in all classes derived from C.* Design constraints are specific to a particular application or library. This particular constraint is common in general-purpose class libraries. For example, the NIH class library [2] contains many functions which must always be overridden if the library is to function correctly.

A company can mandate corporate-wide stylistic constraints and even some implementation constraints on its programmers. The design constraints are more difficult to formalize, because different projects demand different sets of constraints; the constraints may even vary within a single project. At the same time, these design-specific constraints are often the most important to enforce.

Scores of articles, books and informal guidelines have been published on design issues [3, 4, 6, 10]; `comp.lang.c++` and similar Usenet newsgroups host hundreds of gripes from real programmers around the globe on the inability of C++ to specify constraints that would make software design and implementation more of an intellectual activity by cutting down the amount of busy-work. It is clear that the software world needs a tool which will allow software developers to formally express their constraints and will warn them of violations of these constraints. With this goal in mind, we have developed a

This paper was originally presented in September 1993 at the IEEE Conference on Software Maintenance in Montréal, Québec.

new language, CCEL (pronounced “cecil”), the C++ Constraint Expression Language, that allows developers to specify a wide variety of constraints on software systems and to automatically detect violations of these constraints.

Our requirements in designing CCEL were the following:

- It had to offer sufficient *expressive power* to allow programmers to specify a wide variety of constraints, including constraints on both concrete syntax (stylistic constraints) and semantics (design and implementation constraints).
- It had to be relatively *easy to learn and use*. In particular, the syntax and semantics of the language had to mesh well with the syntax and semantics of C++. We chose as our point of departure C++ itself and the well-known assert macro [18] from the standard C library.

In this paper, we demonstrate why CCEL is a powerful tool for building and maintaining complex software systems by expressing a diverse selection of stylistic, implementation and design constraints. We do not discuss the implementation architecture of CCEL in this paper, nor do we provide a comprehensive description of the language. A detailed description of the architecture can be found in [9]. A formal language specification is in [8]. As an aid to reading the CCEL constraints, the appendix at the end of the paper includes the CCEL class hierarchy and a list of CCEL member functions.

2 CCEL Syntax and Semantics

CCEL constraints resemble expressions in the predicate calculus, allowing programmers to make assertions involving existentially or universally quantified variables. In general, a constraint violation is reported if there exists a combination of CCEL variable bindings that causes the assertion to evaluate to false. Consider the constraint that *every base class must have a virtual destructor* [3]. CCEL expresses this constraint as:

```
VirtualDtorInBase(
  Class B;
  Class D | D.is_descendant(B);
  Assert(MemberFunction B::bmf; |
      bmf.name() == "~" + B.name() &&
      bmf.is_virtual());
);
```

An English translation for this constraint is:

For all classes B and D such that D is a descendant of B, it must be true that there exists a member function bmf in class B such that bmf's name is a tilde followed by B's name and bmf is virtual.

The name of the above constraint is VirtualDtorInBase. B and D are universally quantified CCEL variables; bmf is an existentially quantified variable. Class and MemberFunction are CCEL classes, which constitute the type system of CCEL (see the appendix). Variables B and D are of type Class — they range over the classes in the C++ sources being constrained. A consistent binding between B and D is found when the C++ class that D binds to inherits from the C++ class that B binds to. bmf is of type MemberFunction, and its values range over the member functions of the class that B binds to. is_descendant(), name(), is_virtual(), operator==(), operator+() and operator&&() are CCEL class member functions (see the appendix). Function calls to CCEL class member functions are used to construct CCEL expressions. In the above example:

```
D.is_descendant(B)
```

and

```
bmf.name()== "~" + B.name() &&
                 bmf.is_virtual()
```

are CCEL expressions. The "~" is a string constant.

This example demonstrates the five parts of a CCEL constraint (we refer to the above example within parentheses):

1. A **unique identifier** that serves as the name of the constraint (VirtualDtorInBase).
2. A **set of declarations** for universally quantified CCEL variables (B, D). Such variables take as their values components of C++ programs. Each CCEL variable has a type (Class, MemberFunction). CCEL variables may thus range over C++ class names, over C++ function names, over C++ type names, etc.
3. An **assertion** that comprises the essence of the constraint. Assertions may declare existentially quantified variables (bmf).
4. A **scope restriction** that limits the region of applicability of the constraint in relation to the C++ source being checked for constraint violations. By default, constraints are globally applicable, but they may be limited to applying to a single file, function, or class. The example above uses the default global applicability.
5. A **violation message** to be issued when a violation of the constraint is detected. The violation message can be defined by the writer of the constraint. Our example uses the default.

The unique identifier and the assertion are the only two required parts of a CCEL constraint. If any of the other three parts is left out, its default value is assumed.

3 Example Constraints

Readability, reliability and maintainability all require that the meaning of the code be intuitively graspable [4]. C++ is a language full of subtleties. Its programmers impose restrictions on the usage of the language in order to work within the subset known to them to build projects that give predictable and reliable results. CCEL is particularly suitable for expressing these constraints. The scope of this paper does not allow a full discussion of CCEL's functionality. However, this section exposes the flexibility CCEL offers in formally expressing constraints. What follows are not contrived hypothetical examples, but are instead real constraints put forth by real people. The three subsections are devoted to stylistic, implementation and design constraints.

3.1 Stylistic Constraints

Aesthetics is an important consideration for code to be readable, and an essential criterion for code to be maintainable. Hence the large emphasis placed on stylistic constraints by companies and software teams. Most programmers are constrained by a document dictating how to name identifiers — variables, parameters, functions, classes, etc. However, adopting a standard and adhering to it are, in real life, two completely different things. It is very easy to violate one's own naming conventions, and even easier to violate those imposed by someone else. The problem worsens if the standards change to eliminate inconsistencies, to accommodate new developments in the language (e.g. when C++ added templates and exception handling), or to keep up with new whims of management.

CCEL offers a way to formally express many naming conventions, and the CCEL constraint enforcement mechanism [9] flags any violations of these conventions. In the event of a change in naming conventions, all that needs to be changed is the rule specifying a constraint, and everybody is instantly made up-to-date with the changes in the standards.

Example: **Naming Convention for Classes**

Consider the common convention that *all class names must begin with an upper case letter*. CCEL expresses this constraint in the following way:

```
// Class names must begin with a capital letter.
CapitalizeClassName (
  Class C;
  Assert(C.name().matches("^[A-Z]"));
);
```

This constraint has the variable `C` which binds to every class in C++ source code and asserts that the name of the class must match the regular expression in the string in quotes. The syntax and semantics of the regular expression are identical to the ones used for the UNIX utility `grep` [17]. Any class name that fails to match this regular expression will violate the constraint and will thus trigger a violation message.

Example: **Restriction on Variable Names**

The C++ standard [1] specifies that identifiers containing two consecutive underscore characters ("_ _") are reserved for compiler implementations. For this reason, it is wise to avoid using "_ _" in variable names. This leads to the constraint: *No variable may contain "_ _" in its name*. CCEL expresses this constraint as follows:

```
// No variable may contain "_ _" in its name.
NoDblUnderscoreInVarName (
  Variable v;
  Assert(! v.name().matches("__"));
);
```

3.2 Implementation Constraints

Implementation constraints are constraints on legal programs, i.e. on programs that compile and execute but that are likely to display unpredictable and even wrong behavior. These constraints can be applicable to almost all programs and are not specific to particular designs.

Example: **Constructors and Destructors**

The idea behind a constructor is to initialize an object during its creation. Similarly, the idea behind a destructor is to clean up after an object (often including dynamically allocated memory). The existence of an explicitly defined constructor implies that something (most generally, data members) needs to be initialized during creation. It is logical to think that having a destructor is therefore in order, to clean up what has been initialized in the constructor. Similarly, having a destructor without having an explicitly defined constructor is suspect — if there is a need for a destructor, it only makes sense that things to be cleaned up should have been initialized in the constructor. Constructor and destructor declarations should therefore go hand in hand. This is such a common constraint that many compilers generate warnings if it is violated.

CCEL can be used to express this conceptually single constraint as two related constraints as follows:

```
// If a class declares a constructor, it must also declare a
// destructor, and vice-versa.
CtorDtorTogether {
  Class C;
  IfCtorThenDtor (
    MemberFunction C::mfc |
      mfc.name() == C.name();
    Assert(MemberFunction C::mfd; |
        mfd.name() == "~" + C.name());
  );
  IfDtorThenCtor (
    MemberFunction C::mfd |
      mfd.name() == "~" + C.name();
    Assert(MemberFunction C::mfc; |
        mfc.name() == C.name());
  );
};
```

The above constraint introduces the concept of a *constraint class*, used to group a number of logically related constraints. The variables (`C`) inside the constraint class (`CtorDtorTogether`) but outside the individual constraint rules (`IfCtorThenDtor` and `IfDtorThenCtor`) are universally quantified for all the rules inside the constraint class.

Example: Necessary Copy Constructor and Assignment Operator

The *copy constructor* is used to *initialize* an object with an existing object of the same type. The assignment operator is used to *assign* one existing object to another existing object of the same type. In the absence of user-defined copy constructors and assignment operators, C++ generates default functions, which perform memberwise copy or assignment. Any initialized pointer data member in the original object and its copied counterpart will point to the same location in memory. Disasters arising from this situation are almost unavoidable — the common but mistaken notion that we are dealing with copies of the original and the invalidation of the pointed-to object when one of the container objects goes out of scope (its destructor thus deleting what the data member was pointing to) are two examples [7].

For the sake of *correct* program behavior, it is essential that any class with a pointer data member declare a copy constructor and an assignment operator. CCEL can express this constraint:

```
// All classes declaring a pointer member must declare a copy
// constructor and an assignment operator.
NecessaryCopyCtorAssignmentOp {
  Class C;
  DataMember C::m | m.is_pointer();
  PtrDeclaredImpliesCopyCtor (
    Assert(MemberFunction C::mf;
        Parameter mf(p); |
        mf.name() == C.name() &&
        mf.num_params() == 1 &&
        p.is_reference() &&
        p.is_const() &&
        p.type().basic_type() == C);
  );
  PtrDeclaredImpliesAssignmentOp (
    Assert(MemberFunction C::mf;
        Parameter mf(p); |
        mf.name() == "operator=" &&
        mf.num_params() == 1 &&
        p.is_reference() &&
        p.is_const() &&
        p.type().basic_type() == C);
  );
};
```

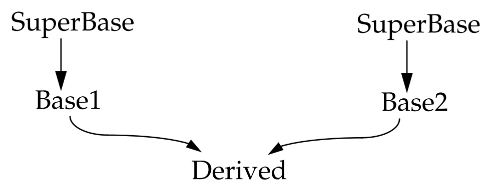
An important observation here is that CCEL needs no concept of what a constructor or what an assignment operator is. Yet, it is capable of building up these high-level concepts from simple building- blocks like names and types of identifiers, number of parameters, etc. This is representative of the power and flexibility of CCEL — it is powerful, because it can refer to high-level concepts and represent them; it is flexible, because the small building blocks can build a wide variety of complex expressions required to formalize a constraint.

Example: Public Inheritance Virtual

C++ supports multiple inheritance: a class can be derived from more than one base class. With this ability comes the possibility that a derived class may have more than one path to a base class. Unless all such paths are virtual, multiple instances of the base class will exist. For example,

```
class SuperBase {...};
class Base1 : public SuperBase {...};
class Base2 : public SuperBase {...};
Class Derived : public Base1, public Base2
{...};
```

A `Derived` object will look like this:



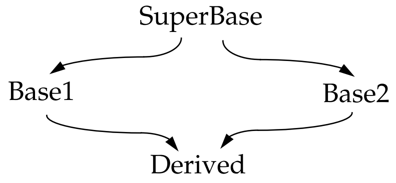
Ambiguous — two copies of `SuperBase`

Ambiguities can arise with regard to which copy of the `SuperBase` one is referring to. There are work-arounds to ambiguities, but adopting them is usually error-prone and confusing. This almost always leads to incorrect semantics and unpredictable program behavior.

Markku Sakkinen argues [11] that public inheritance is basically an *IsA* relationship — if class D is derived from class B, D *is a* B. The important point is that D is a B only *once*; it does not make sense to say that D is B more than once. The above hierarchy represents exactly this "nonsensical" thing. Declaring the public inheritance between `SuperBase` and each of `Base1` and `Base2` virtual prevents this ambiguity:

```
class Base1 : public virtual SuperBase {...};
class Base2 : public virtual SuperBase {...};
```

Now a `Derived` object looks like:



Unambiguous — only one copy of `SuperBase`

The following CCEL constraint can be used to ensure that nobody falls into this “nonsensical” trap:

```
// All public inheritance must be virtual.
PublicInheritanceVirtual (
  Class B;
  Class D | D.is_child(B) &&
    D.is_public_descendant(B);
  Assert(D.is_virtual_descendant(B));
);
```

This constraint actually forces *all* public inheritance (not only cases of multiple inheritance) to be virtual. This is, in fact, what Sakkinen advocates. Following Sakkinen’s advice, however, often introduces additional overhead because of the way C++ compilers typically implement virtual base classes [1]. A preferable constraint may therefore be: *if there are multiple public inheritance paths from a derived class to one of its base classes, all these paths must be virtual.* CCEL can be used to express this selective constraint as well:

```
// If there are multiple public inheritance paths from a derived
// class to one of its base classes, all these paths must be
// virtual.
MultiPubInheritanceMustBeVirtual (
  Class C;
  Class D | D.is_child(C) &&
    D.is_public_descendant(C);
  Class E | E != D && E.is_child(C) &&
    E.is_public_descendant(C);
  Class F |
    (F == D || F.is_public_descendant(D)) &&
    (F == E || F.is_public_descendant(E));
  Assert(D.is_virtual_descendant(C) &&
      E.is_virtual_descendant(C));
);
```

Note that the term *child* refers to a direct descendant.

Example: Return Type of Assignment Operator

C++ experts agree on the usefulness of the assignment operator. However, they do not agree on the *return type* of that operator. For example, Meyers argues [3] that the return type should be a reference to the class so that assignment chaining is possible, since this is a supported feature of built-in types:

```
a = b = c     // this is not possible with return type void
```

CCEL lets the followers of this convention write a constraint to check for violations by the classes they use:

```
// The return type of operator= must be a reference to the class.
ReturnTypeOfAssignmentOpIsRef (
  Class C;
  MemberFunction C::mf |
    mf.name() == "operator=";
  Assert(mf.is_reference() &&
      mf.type().basic_type() == C);
);
```

Note that in CCEL, unlike in C++, the type of a function is its return type.

Rob Murray sees a problem with the above convention. His argument is based on the questionable utility of the following statement [12]:

```
(a = b) = c
```

which, he argues, is an unintuitive and confusing assignment. He proposes a more strict constraint by saying that the return type should be a *const* reference to the class. CCEL can be used to express this constraint, too:

```
// The return type of operator= must be a const reference to
// the class.
ReturnTypeOfAssignmentOpIsConstRef (
  Class C;
  MemberFunction C::mf |
    mf.name() == "operator=";
  Assert(mf.is_const() &&
      mf.is_reference() &&
      mf.type().basic_type() == C);
);
```

On the other hand, Doug Lea points out that subtle interactions between the strong static typing of C++ and its support for polymorphism can sometimes lead to unanticipated resolution of calls to overloaded functions, including `operator=` [13]. These problems, the details of which are unimportant here, can be avoided by having `operator=` return `void`. CCEL can be used to express this constraint as well:

```
// The return type of operator= must be void.
ReturnTypeOfAssignmentOpIsVoid (
  Class C;
  MemberFunction C::mf |
    mf.name() == "operator=";
  Assert(mf.type().name() == "void");
);
```

These three examples are indicative of the flexibility of CCEL. CCEL does not say what constraint is correct, but it allows software developers to enforce the constraints they believe in.

Example: Restriction on Template Arguments

In C++, class templates can take either type or non-type parameters, but function templates can take only type parameters. This inconsistency can make it impossible to write global functions to work with template classes, including input and output operators. Moreover, allowing non-type parameters to templates often causes "code bloat" in the following way:

```
template<class T, int size> class Array {
  private:
    T array[size];
    //...
};
```

This template creates a *different* class for each combination of `T` and `size` — potentially a lot of code duplication. Both these problems have led some to argue for this constraint: *disallow non-type arguments to templates*. This constraint can be expressed in CCEL as follows:

```
// Disallow non-type arguments to templates.
NoNonTypeArgToTemplate (
  Template T;
  Parameter T<p>;
  Assert(FALSE);
);
```

CCEL differentiates between type parameters and non-type parameters, called `TypeParameter` and `Parameter`, respectively. The above rule issues violation messages whenever there is a non-type parameter to a template class.

CCEL can also be used with templates to approximate Meyers' notion of *constrained genericity* [14], i.e. the ability to restrict template type arguments to those supporting a specific set of functions.

3.3 Design Constraints

So far we have shown constraints that are difficult to remember without an automated violation checker. Nevertheless, they are generic enough so that they could be part of a written coding standards document. They are *design-independent* constraints. However, it is *design-dependent* constraints that are the most important in practice. At the same time, they are the hardest to express, because of the absence of a suitable language. This is where the expressive power and flexibility of CCEL comes in. CCEL excels in formalizing a wide selection of design constraints ranging from the easiest, most intuitive to the most obscure. The following rules, each of which is taken from real C++ systems, provide an indication of what CCEL can do for real projects.

Example: Required Base Class

One effective C++ library design entails putting together classes which are compatible with each other and which share some consistent behavior. Maintaining a library and extending it involves knowledge of the commonality of its components. One way to enforce consistency and compatibility among the classes is to encapsulate all the commonality into a root class from which all the other classes inherit. The NIH Class Library [2] defines such a root class called `Object` and makes every class a descendant of it.[1] Class `Object` declares virtual functions that apply to all classes — functions for copying, printing, storing, reading and comparing objects, for example. The viability of the library is *fundamentally dependent* on the fact that each class has a path to the root in the inheritance hierarchy; if this constraint is violated, disaster typically results.

CCEL can be used to make sure that whoever is extending the library respects the rule: *make all new classes inherit from* `Object`:

```
// Every class must inherit from the class Object.
MustInheritFromObject (
  Class B | B.name() == "Object";
  Class C | C != B;
  Assert(C.is_descendant(B));
);
```

Note that `B` can only be bound to one class.

Example: Required Member Function

In C++, the constructors for a class initialize objects of that type. Since the constructors do not have a return value, it is difficult to determine whether the initialization of the object was successful. Similarly, there is no built-in way to determine whether an existing object is in a valid state. To address this shortcoming, a fundamental design constraint in the GNU C++ class library, libg++ [15], is that every object must define a function indicating its state. Most of the libg++ classes define a method named `OK` which checks the *representation invariant* of a class object. This is a (sometimes partial) verification of the library's promise that (1) class operations always leave objects in valid states, and (2) the class protects itself so that client functions cannot corrupt this state. `OK` checks the object's status by looking at some of its private representational properties. No simple validation method will guarantee that all operations perform correctly, but `OK` can at least verify that operations do not corrupt representations.

To ensure that classes within libg++ adhere to the convention of offering an `OK` function, CCEL can be employed as follows:

```
// Every class must declare a function called OK.
MustDefineOK (
  Class C;
  Assert(MemberFunction C::mf; |
      mf.name() == "OK");
);
```

Example: Required Function Overriding

That C++ constructors cannot be virtual is a much discussed topic, the reason being that the constructor needs to know the exact type of object it is creating. Sometimes,

1. Actually, a few classes in the library fail to follow this constraint, but for our purposes here the exceptions are unimportant.

however, it is convenient to be able to construct a new object based on another object's dynamic type. This can be done by simulating the effect of a *virtual constructor* by a virtual function `clone`. In general, such a function creates a new object on the free store and returns a pointer to it. All derived classes must override the `clone` function to return an object of the appropriate type. However, disaster results if a derived class does *not* override the `clone` function: a call to `clone()` for the derived class results in calling the base level `clone` function, and an object of the base type is returned. This error is detected at neither compile time nor run time, but manifests itself only through incorrect program behavior.

For applications using virtual constructors, it is crucial to constrain every class to override the function `clone`. There is no way to enforce this constraint in C++, but CCEL can be used to do it:

```
// The virtual function clone must be overridden in any class
// inheriting from the class Object.
CloneInClassObjectOverriddenInDescendant (
  Class B | B.name() == "Object";
  Class D | D.is_descendant(B);
  MemberFunction B::bmf |
    bmf.name() == "clone" &&
    bmf.is_virtual();
  Assert(MemberFunction D::dmf; |
      dmf.overrides(bmf));
);
```

Example: Derivation Prohibition

Sometimes a programmer has to resort to obscure tricks in C++ in order to create a work-around for a problem, because the raw language lacks some necessary functionality. More often than not, these work-arounds require a deep understanding of the language and result in the use of ad hoc methods and the implementation of unintuitive code. This is clearly a deterrent to producing readable and maintainable code. A recent thread in the Usenet newsgroup comp.lang.c++ posed the constraint question: *How does one prevent derivation from a particular class?*

For a given class `LeaveMeAlone`, this constraint can actually be implemented in C++, but the solution is far from intuitive:

```
class LeaveMeAloneLock {
    friend LeaveMeAlone;
  private:
    LeaveMeAloneLock() {}
};
class LeaveMeAlone: public virtual
                      LeaveMeAloneLock {
  public:
    LeaveMeAlone();
    // ...
};
```

This solution involves sophisticated knowledge of C++ subtleties. Several pieces of information are needed to realize how this code works. Because `LeaveMeAloneLock` is a virtual base of `LeaveMeAlone`, any class inheriting from `LeaveMeAlone`, when instantiated, will have to initialize `LeaveMeAloneLock`. An implicit call is made to `LeaveMeAloneLock`'s default constructor, which, being a private constructor, cannot be accessed by any class other than the `friend` class `LeaveMeAlone`. Hence, it is impossible to derive from class `LeaveMeAlone`. The error is detected where a constructor is defined or generated for the improperly derived class.

An important observation about the above C++ solution is that not only does one need a great deal of information to formulate it, but each piece of information is *critical* for the solution to work. The fact that `LeaveMeAloneLock` has a private default constructor, that `LeaveMeAloneLock` declares `LeaveMeAlone` as a friend, that `LeaveMeAloneLock` is a virtual base of `LeaveMeAlone`, etc. If one piece of this chain is broken, so is the solution. This concentration of unintuitive code makes the solution especially fragile.

In contrast, CCEL can be used to write a simple and straightforward constraint to solve the problem:

```
// No classes should inherit from the class LeaveMeAlone.
NoDerivation (
  Class C | C.name() == "LeaveMeAlone";
  Class D;
  Assert(! D.is_descendant(C));
);
```

Example: Restriction on Template Instantiation

Another recent posting in comp.lang.c++ addressed the need for "built-in listability" for some classes. We have chosen to include it here to show how CCEL can be used to express a particularly difficult constraint. The constraint is that *any type used to instantiate the* `List` *template class must be descended from the* `List` *template class itself.* That is:

```
template<class T> class List { ... };
class A : public List<A> { ... };
class B { ... };
List<A> a_list;              // okay
List<B> b_list;              // not okay
```

This is a complicated constraint. It is hard to understand, even harder to write down. It is also difficult to remember, a fact which makes it less likely to be followed by everyone on a team. All these are compelling reasons to formally and automatically enforce this kind of rule. With CCEL, one can easily maintain a formal definition of the rule:

```
// All types T used to instantiate the List template must be
// descended from List<T>.
ProperUseOfListBoltOn (
  Template T | T.name() == "List";
```

```
    TypeParameter T<P>;
    Class C |
      C.name() == "List" + "<" + P.name() + ">";
    Assert(P.is_descendant(C));
  );
```

Here CCEL makes use of the name of the template instantiation to give a succinct, flexible and powerful constraint.

4 Engineering Considerations

By default, CCEL constraints apply to all source code in the system being worked on. It may sometimes be necessary to restrict the scope of applicability of a constraint to a certain file, function or even a class. For example, it is possible that a programmer might use one set of naming conventions for a class library and another set of naming conventions for application-specific classes. CCEL addresses this need explicitly. For example, the constraint *the names of all classes in file "CapClasses.H" must begin with an upper case letter* can be written in CCEL using one of the rules we have already seen, this time using a file scope:

```
  // Capitalize class names only in file "CapClasses.H".
  File "CapClasses.H" : CapitalizeClassName (
    Class C;
    Assert(C.name().matches("^[A-Z]"));
  );
```

On the other hand, it is sometimes more convenient to *disable* a global constraint for a specific scope. CCEL allows this by creating a new constraint which disables an old constraint for the specified scope. Consider the constraint: *The constraint CapitalizeClassName must not apply to class lowercaseClass:*

```
  // The constraint CapitalizeClassNames must not apply to
  // the class lowercaseClass.
  Class "^lowercaseClass$" : NoCapClassName (
    Disable CapitalizeClassName;
  );
```

Note that file scopes are specified using UNIX shell wildcard expressions, but the function and class scopes are specified using UNIX regular expressions [17]. In the last example, the UNIX wildcard characters `^` and `$` are used to indicate the beginning and end of a string representing the class name.

Similarly, an `Enable` feature exists to *enable* a constraint for a given scope. Details can be found in [8]. It is possible to attain an arbitrary level of nested applicability of several constraints by using the scope, enable and disable features of CCEL constraints.

5 Application to Other Languages

The details of CCEL are clearly specific to C++. The fundamental design principles behind CCEL, however, apply equally well to other languages. The kinds of constraints described in this paper exist for languages like Smalltalk and CLOS as much as they do for C++ [16]. The desirability of choosing a syntax, semantics, and conceptual model that is familiar to programmers is as important for an Eiffel constraint language as it is for CCEL. The software engineering considerations that allow CCEL constraints to be bundled into constraint classes, to be explicitly disabled, and to have user-specified scopes and violation messages are as important for Object Pascal programmers as they are for C++ software developers. Furthermore, the software architecture behind a CCEL implementation [9] contains nothing that tailors it to the idiosyncrasies of C++. In short, the primary design considerations — both in terms of the language itself and in terms of the implementation of that language — are divorced from the specifics of C++ and can be directly applied to constraint languages for other programming languages.

6 Related Work

Support for formal design constraints in the form of assertions or annotations was designed into Eiffel [19], has been grafted onto Ada in the language Anna [20], and has been proposed for C++ in the form of A++ [21, 22]. This work, however, has grown out of the theory of abstract data types [23], and has tended to limit itself to formally specifying the semantics of individual functions and/or collections of functions (e.g., how the member functions within a class relate to one another). In general, violations of these kinds of constraints can only be detected at runtime. Our work on CCEL has a different focus. We are interested in constraints whose violations can be detected at compile time, and we are further interested in addressing the need to constrain relationships between classes, which Eiffel, A++, and Anna are unable to do. CCEL can also express constraints on the concrete syntax of C++ source code (e.g., CCEL class-specific naming conventions); this is also outside the purview of semantics-based constraint systems.

The commercial product CodeCheck [24] allows for the specification of user-defined constraints in a similar spirit as CCEL, but the model behind CodeCheck is procedural, not object-oriented; this is a reflection of the fact that CodeCheck was originally designed to express constraints on C programs. In addition, CodeCheck constraints are primarily procedural (`if-then-else` constructs), whereas CCEL constraints are primarily declarative.

Formal specification languages such as Z [25], VDM [25], and Larch [26] are designed to allow developers to specify the behavior of software systems independent of the language(s) used to implement the systems. Such specification languages can express a wide variety of powerful constraints, but their semantic model and syntactic appearance is unfamiliar to most practicing programmers, and as a result they have not seen wide adoption. CCEL, though less expressive than general-purpose specification languages, is at once simpler and more intuitive for C++ programmers, and we believe this will lead to its being more readily embraced by the development community.

7 Status and Future Directions

A prototype version of CCEL exists and can process constraints and detect violations in C++ source code. A library of CCEL constraints inspired by the plethora of stylistic, implementation and design constraints [3, 4, 6, 10, 14] is currently being compiled.

CCEL currently supports the expression of constraints on C++ *declarations* only. Our plan is to extend CCEL into the next-generation CCEL-II, which will add the ability to express constraints on C++ *definitions* also. This will greatly multiply the power, flexibility and scope of CCEL and will enhance the utility of this tool for the practical world of software designers and developers.

8 Acknowledgments

Carolyn K. Duby and Yueh-hong Lin were major contributors to the design and development of CCEL. Special thanks to Lalit K. Agarwal, Vidur Apparao, Maarten van Dantzich, Tony Davis, Robert Duvall, Moises Lejter, Yueh-hong Lin, Steve Reiss and Supriya Wickrematillake for their helpful comments on earlier drafts of this paper. We are grateful to an anonymous referee for bringing to our attention some related work which we had originally overlooked.

Partial support for this research was provided by the NSF under grants CCR 9111507 and CCR 9113226, by ARPA order 8225, and by ONR grant N00014-91-J-4052.

9 References

[1] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

[2] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.

[3] Scott Meyers. *Effective C++: 50 Ways to Improve Your Programs and Designs.* Addison Wesley, 1992.

[4] Thomas Plum and Dan Saks. *C++ Programming Guidelines.* Plum Hall Inc., 1991.

[5] Bjarne Stroustrup. *The C++ Programming Language, Second Edition.* Addison Wesley Publishing Company, 1991.

[6] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley Publishing Company, 1992.

[7] Scott Meyers and Moises Lejter. "Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++". *Usenix C++ Conference Proceedings*, Spring 1991.

[8] Yeuh-hong Lin and Scott Meyers. "CCEL: The C++ Constraint Expression Language." Technical report 93-23, Brown University Department of Computer Science, April 1993.

[9] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. "Constraining the Structure and Style of Object-Oriented Programs". *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, April 1993.

[10] Mats Henricson and Erik Nyquist. *Programming in C++: Rules and Recommendations.* Ellemtel Telecommunications Systems Laboratories, 1990.

[11] Markku Sakkinen. "A Critique of the Inheritance Principles of C++". *Computing Systems*, Vol. 5 No. 1, Winter 1992.

[12] Robert Murray. "The C++ Puzzle". *The C++ Report*, Vol. 2 No. 10, November/December 1990.

[13] Doug Lea, personal communication, 1993.

[14] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[15] libg++ documentation. Gnu Library. Free Software Foundation, June 1991.

[16] Gregor Kiczales and John Lamping. "Issues in the Design and Implementation of Class Libraries". *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92)* (Andreas Paepcke, ed.), October 1992.

[17] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, 1984.

[18] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988.

[19] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, 1992.

[20] D. Luckham, F. von Henke, B. Krieg-Bruckner, and O. Owe, "Anna, A Language for Annotating Ada Programs: Reference Manual", *Lecture Notes in Computer Science* 260, Springer-Verlag, 1987.

[21] Marshall P. Cline and Doug Lea, "Using Annotated C++", *Proceedings of C++ at Work - '90*, September 1990.

[22] Marshall P. Cline and Doug Lea, "The Behavior of C++ Classes," *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

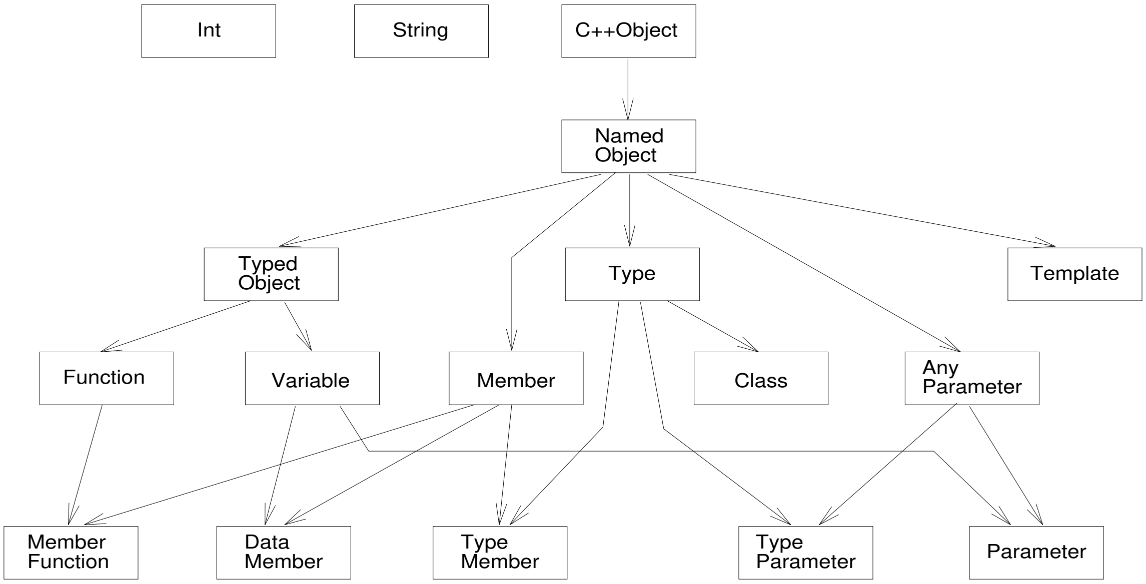
[23] Barbara Liskov and John Guttag, *Abstraction and Specification in Program Development*, MIT Press, 1986.

[24] Loren Cobb, *C Code Analysis Using Codecheck*, Abraxas Software, 1992.

[25] *VDM '90: VDM and Z: Formal Methods in Software Development: Third International Symposium of VDM Europe*, Kiel, Germany, April 17-21, 1990.

[26] John V. Guttag, James J. Horning, and Jeannette M. Wing, "The Larch Family of Specification Languages," *IEEE Software*, September 1985.

Appendix



CCEL class hierarchy

CCEL Class	Member Function
Int	Int operator == (Int) Int operator < (Int) Int operator > (Int) Int operator <= (Int) Int operator >= (Int) Int operator ! (Int) Int operator != (Int) Int operator && (Int) Int operator || (Int)
String	Int operator == (String) Int operator < (String) Int operator > (String) Int operator <= (String) Int operator >= (String) Int operator != (String) String operator + (String) Int matches(String)
C++Object	String file() Int begin_line() Int end_line()
NamedObject	String name() Int scope_is_global() Int scope_is_file()
TypedObject	Type type() Int num_indirections() Int is_reference() Int is_static() Int is_volatile() Int is_const() Int is_array() Int is_long() Int is_short() Int is_signed() Int is_unsigned() Int is_pointer()

CCEL Class	MemberFunction
Type	Int has_name(String) Type basic_type() Int operator == (Type) Int operator != (Type) Int is_convertible_to(Type) Int is_enum() Int is_class() Int is_struct() Int is_union() Int is_friend(Class) Int is_child(Class) Int is_descendant(Class) Int is_virtual_descendant(Class) Int is_public_descendant(Class)
Template	Int is_class_template() Int is_function_template()
Function	Int is_inline() Int is_friend(Class) Int num_params()
Member	Int is_private() Int is_protected() Int is_public()
TypeMember	
Variable	Int scope_is_local()
AnyParameter	Int positions()
Class	
MemberFunction	Int is_virtual() Int is_pure_virtual() Int overrides(MemberFunction)
DataMember	
Parameter	Int has_default_value()
TypeParameter	

CCEL class member functions