BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M12

# ``CCEL: A Metalanguage for C++

by

Carolyn Kay Duby

# CCEL : A Metalanguage for C++

Carolyn Kay Duby

Brown University, Providence, RI 02912
Cadre Technologies, Inc., Providence, RI 02903

# CCEL : A Metalanguage for C++

Carolyn Kay Duby

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science in the
Department of Computer Science at Brown University

May 1, 1992

This research project by Carolyn Kay Duby is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

[illegible]

Professor Steven P. Reiss
Advisor

5/1/92

Date

## Abstract

C++ is an expressive language, but it does not allow software developers to say all the things about their systems that they need to be able to say. In particular, C++ offers no way to express many important constraints on a system's design, implementation, and presentation. In this paper, we describe CCEL, a metalanguage for C++ that allows software developers to express constraints on C++ designs and implementations, and we describe Clean++, a system that checks C++ code for violations of CCEL constraints. CCEL is designed for practical, real-world use, and the examples in this paper demonstrate its power and flexibility.

## 1 Introduction

C++ is an expressive language, but it does not allow software developers to say all the things about their systems that they need to be able to say. In particular, C++ offers no way to express many important constraints on a system's design, implementation, and stylistic conventions. Consider the following sample constraints, none of which can be expressed in C++:

- **Design Constraint:** *The member function M in class C must be redefined in all classes derived from C. This applies to both direct and indirect subclasses, so declaring M as a pure virtual function in C does not satisfactorily enforce the constraint.* This kind of constraint is common in general-purpose class libraries. For example, NIHCL [4] contains many such functions for the top-level Object class.
- **Implementation Constraint:** *If a class declares a pointer member, it must also declare an assignment operator and a copy constructor.* Failure to adhere to this constraint almost always leads to incorrect program behavior [12, Item 11]. A number of similar constraints was presented at last year's USENIX C++ conference [13].
- **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 are irritating at best, confusing and misleading at worst.

Constraints such as these exist in virtually every system implemented in C++, but different systems require very different sets of constraints. As a result, it is unreasonable to ask that C++ compilers be augmented to handle these issues. Yet the issues remain, and their importance cannot be ignored. In this paper, we describe CCEL ("Cecil") - the C++ Constraint Expression Language - a metalanguage for C++ that allows software developers to express a wide variety of constraints on C++ designs and implementations, and we describe Clean++, a system that checks C++ code for violations of CCEL constraints.

We took as our original inspiration the lint tool, which reports a number of likely error conditions in C programs. However, the errors C programmers need to detect are qualitatively different from the errors that C++ programmers need to detect. lint concentrates on type mismatches and data-flow anomalies, but type mismatches are not an issue in C++ because the language is strongly typed, and data flow analysis is unrelated to the high-level perspective encouraged by the modular constructs of C++. C++ programmers are concerned with higher-level concepts such as the structure of an inheritance hierarchy. Detection of errors in the inheritance hierarchy requires a tool that provides users with a way to check for *programmer-defined* constraints.

Other important differences between the philosophy behind lint and that behind Clean++ are those of customizability and extensibility. The set of conditions detected by lint cannot be extended by programmers, nor is there an easy way to disable the detection of classes of errors for *parts* of source files. These are significant drawbacks, and both are overcome by CCEL, as the examples in the remainder of this paper will show.

## 2 The CCEL Language

The requirements for a good constraint language are:

- The language must be powerful enough to express the constraints important to the programmer.
- The language must be intuitive and simple to learn. The look and feel must be familiar to the programmer to facilitate learning and use. Programmers need to be able to read a constraint and understand what it means in order to be able to correct a violation of a constraint, to write new constraints, and to modify existing constraints.

CCEL is based on an object-oriented model where metaclasses represent the concepts of C++. The metaclasses are arranged in a multiple inheritance hierarchy (See Figure 1) and have member functions defined for them (See Table 1). We determined the metaclasses and their positions in the hierarchy by first examining in detail the concepts important to C++ programmers and the constraints they need to express. Then we classified the concepts into *metaclasses*, such as C++ classes and member functions, and others into *properties* of metaclasses, such as the protection level of a member function. We determined the metaclass hierarchy by analyzing the features the metaclasses have in common. We added abstract metaclasses such as NamedObject to represent the common features. We chose the object-oriented model because it is familiar to users of C++, and because we can extend the model to add either new member functions or metaclasses as new concepts need to be introduced.

While abstracting the concepts of C++ into CCEL metaclasses, we often had to decide if a concept was a new metaclass or if it could be expressed as a member function of an existing metaclass. For example, the only difference between a class and a struct is that the default protection for a class is private, while the default protection for a struct is public. One possibility would be to put classes and structs in the same metaclass with a boolean member function indicating whether the metaclass is a struct. A second possibility is to put classes and structs in two different metaclasses, with their common functionality abstracted to a base metaclass. In general, we combined concepts into one metaclass when the differences were trivial and the additional complexity of having a new metaclass outweighed the increased functionality.

For example, we divided classes and structs into two metaclasses because C++ programmers often wish to draw a distinction between them. In particular, many users believe that structs should be "just like C," while classes should be used whenever C++-specific features are employed. By separating the metaclass concepts of classes and structs, it is straightforward to write CCEL rules that restrict the features that can be used inside structs. On the other hand, we have not yet encountered a compelling reason for differentiating between functions in general and global functions in particular (as opposed to member functions), to the current CCEL metaclass hierarchy has no metaclass specifically devoted to global functions. This means that there is no way to write a CCEL rule that applies only to global functions, but it would be simple enough to modify the metaclass hierarchy if it were shown to be necessary.

CCEL constraints resemble expressions in predicate calculus, allowing the programmer to make assertions involving existentially or universally quantified metavariables. Clean++ reports any combination of metavariable values that cause the assertion to evaluate to false.

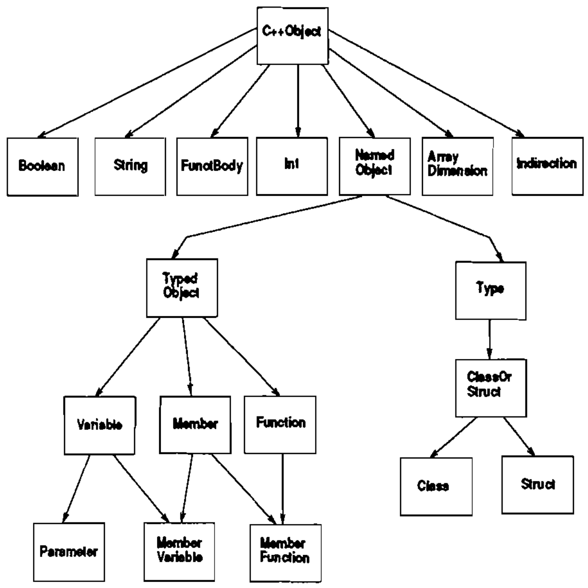

Figure 1: CCEL Metaclass Hierarchy

Each constraint contains an assertion which must be met by some C++ source code. For example, a constraint requiring that all class names begin with a capital letter can be written in CCEL as follows:

```
// Every class name must begin with a capital letter
CapitalizeClassNames (
     Class C;      // C is a class

     Assert(C.name().matches("^[A-Z]"));
);
```

`CapitalizeClassNames` is the identifier which is used to refer to the constraint. As we will see later, this identifier can be used to enable or disable the constraint. `C` is a metavariable whose domain is the set of all C++ classes in the system. The body of the `CapitalizeClassNames` constraint takes the form of an Assert expression, modeled loosely on the standard C assert macro facility. The assertion is that the string representing the name of the class must match the UNIX regular expression "^[A-Z]". **Class** and **Assert** are CCEL keywords; a complete list of keywords can be gleaned from the lex summary in appendix C.

As in C++, all metavariables must be declared before use. They are assumed to be universally quantified unless explicitly existentially quantified. Existential quantification is indicated by the use of square brackets, [...]. For example, in the constraint

```
// Every base class must have a virtual destructor.
VirtualDestInBases (
    Class B, D;

    if (D.is_descendant(B))
        Assert([MemberFunction B::f1; |
               ((f1.name() == "~{B.name()}") && (f1.is_virtual())}]);
);
```

| Metaclass Name | Metaclass Member Functions |
|---|---|
| ArrayDimension | Int value() |
| Boolean | Boolean operator&&(Boolean) |
| | Boolean operator\|\|(Boolean) |
| | Boolean operator!() |
| | Boolean operator==(Boolean) |
| | Boolean operator!=(Boolean) |
| C++Object | Int begin_line() |
| | Int end_line() |
| | String file() |
| Class | |
| ClassOrStruct | Boolean is_descendant(Class) |
| | Boolean is_virtual_descendant(Class) |
| | Boolean is_public_descendant(Class) |
| | Boolean is_friend(Class) |
| FunctBody | Boolean calls(String) |
| Function | Int num_params() |
| | Boolean is_inline() |
| | FunctBody body() |
| | Boolean is_friend(Class) |
| Indirection | Int level() |
| | Boolean is_const() |
| Int | Boolean operator==(Int) |
| | Boolean operator<=(Int) |
| | Boolean operator>=(Int) |
| | Boolean operator<(Int) |
| | Boolean operator>(Int) |
| | Boolean operator!=(Int) |
| Member | Boolean is_private() |
| | Boolean is_protected() |
| | Boolean is_public() |
| MemberFunction | Boolean is_virtual() |
| | Boolean is_pure_virtual() |
| | Boolean redefines(MemberFunction) |
| MemberVariable | |
| NamedObject | String name() |
| Parameter | Int position() |
| | Boolean has_default_value() |
| String | Boolean operator==(String) |
| | Boolean operator<=(String) |
| | Boolean operator>=(String) |
| | Boolean operator<(String) |
| | Boolean operator>(String) |
| | Boolean operator!=(String) |
| | Boolean matches(String) |
| Struct | |
| Type | Boolean convertible_to(Type) |
| | Boolean operator==(Type) |
| | Boolean is_enum() |
| | Boolean is_union() |
| TypedObject | Int num_indirections() |
| | Boolean is_pointer() |
| | Boolean is_static() |
| | Boolean is_reference() |
| | Boolean is_volatile() |
| | Boolean is_const() |
| | Boolean is_array() |
| | Boolean is_long() |
| | Boolean is_short() |
| | Boolean is_signed() |
| | Boolean is_unsigned() |
| | Type type() |
| Variable | Boolean scope_is_local() |
| | Boolean scope_is_file() |
| | Boolean scope_is_class() |

Table 1: CCEL Metaclass Member Functions

the metavariables `B` and `D` are universally quantified, while the metavariable `f1` is existentially quantified. In English, this constraint reads, "For all classes `B` and `D`, if `D` is a descendant of `B`, then it must be true that there exists a member function `f1` in `B` such that `f1`'s name is a tilde followed by `B`'s name, and `f1` is virtual." The legal types for metavariables are the CCEL metaclasses shown in Figure 1.

Metavariable declarations may have a condition attached to them, which is indicated by a vertical bar and a boolean expression following the variable name. For example,

```
Class B;                        //domain is all classes
Class D | (D.is_descendant(B)); //domain is only classes derived from B
```

means, "for every class `B` and every class `D` such that `D` is a descendant of `B`".

By default, a constraint applies to all code in the system. This is not always desirable. For example, consider the case where a programmer has a set of naming conventions for a class library that differ from the naming conventions used for application classes. The ability to enable and disable constraint checking for named parts of the system is an important feature of CCEL. For example, if we wanted to limit the applicability of `CapitalizeClassNames` to the file "objects.C", we could declare a scope for the constraint as follows:

```
// For every class C in file "objects.C", the class name must match the
// UNIX regular expression ^[A-Z].
File "objects.C" : CapitalizeClassNames (
    Class C;

    Assert(C.name().matches("^[A-Z]"));
);
```

Sometimes it is more convenient to specify where an otherwise global constraint does *not* apply. If `CapitalizeClassNames` applies to every C++ class except `example`, we could `disable` `CapitalizeClassNames` for that class as follows:

```
// Do not report violations of CapitalizeClassNames in C++ class
// "lowercaseNames".
Class lowercaseNames : DontCapitalizeInLowercaseNames (
    disable CapitalizeClassNames;
);
```

Individual constraints may be grouped together into constraint classes. Suppose there are several constraints enforcing naming conventions. They could be grouped together in a constraint class called `NamingConventions` as follows:

```
ConstraintClass NamingConventions {
  // For every class C, the class name must match the UNIX regular expression
  // ^[A-Z].
  CapitalizeClassNames
  (
      Class C;

      Assert(C.name().matches("^[A-Z]"));
  );

  // For every function F, the function name must begin with
  // a lower case letter.
  SmallFunctNames
  (
      Function F;

      Assert(F.name().matches("^[a-z]"));
  );
};
```

Notice that constraint classes are demarcated by brackets {...}, while individual constraints use parentheses (...). Constraint classes may be disabled by having a constraint such as this:

```
NamingConventionsOff (
   disable NamingConventions;
);
```

Like all CCEL constraints, this one is implicitly globally applicable. A particular constraint in a constraint class can be disabled by using the C++ scoping operator("::"):

```
SomeNamingConventionsOff (
    disable NamingConventions::CapitalizeClassNames;
);
```

If the assertion condition for a constraint is complex, the constraint designer may want to create two or more simpler constraints. The following example is a set of constraints that reports undeclared assignment operators for classes that contain a pointer member or are derived from a class containing a pointer member:

```
// If a class contains a pointer member, it must declare an assignment operator.
AssignmentMustBeDeclaredCond1 (
    Class          C;
    MemberVariable C::v;  // v is a member variable of C

    if (v.is_pointer())
        Assert([MemberFunction C::f; |
               (f.name() == "operator=")]);
);

// If a class inherits from a class containing a pointer member, the derived
// class must declare an assignment operator.
AssignmentMustBeDeclaredCond2 (
    Class B;
    Class D | D.is_descendant(B);
    MemberVariable B::bv;  // bv is a member variable of class B

    if (bv.is_pointer())
        Assert([MemberFunction D::df; |
                   (df.name() == "operator=")]);
);
```

The ":  :" notation in the metavariable declaration of v in the first constraint indicates that v is a metavariable whose domain is the member variables of class C. To be a member of a class means that a member variable or member function is declared in this class, i.e. is not inherited. The `redefines` member function of the `Class` metaclass (see Figure 1) can be used to find out if a class defines a function with the same parameters and name as the given member function, i.e. if C++ would view it as a virtual redefinition. In the first constraint, the existentially quantified metavariable `f` is used to determine if an assignment operator is defined for classes containing a pointer member variable. In the second constraint, the existentially quantified variable `df` is used to check if the descendants of a base class containing a pointer member define an assignment operator.

The `Member`, `MemberVariable`, and `Parameter` metaclasses have special conditions attached to them similar to the `MemberFunction` object. A `::` in a `MemberVariable` or `Member` variable declaration separates the class name from the member variable name. Examples:

```
  Class C;                    // C is a class
  Member C::M1;               // M1 is a member of class C
  Function      F;            // F is a function
  Parameter     F(P);         // P is a parameter of F
  Variable      V;            // V is a variable with local, file, or class scope
```

Any combination of the C++ relational and logical operators can be used inside `Assert` clauses. As with C++, only types for which comparison has been defined can be used in comparison operations. The types of the items being compared must be the same, and the arguments to the boolean predicates must be boolean expressions.

## 3 Error Messages

Since constraints are user-defined and not hardcoded, the CCEL evaluator cannot form a meaningful error message describing what the error condition is. The best the evaluator can do is print the constraint violated and the current values of the metavariables that violated the assertion. Therefore, we decided to allow the user to optionally associate a message to be reported with every constraint. The programmer-defined error messages enables the user to word the error message in familiar terms and also allows users who do not know CCEL to use Clean++ to find program errors. If CCEL did not allow users to define their own error messages, the only way for a user to determine the error condition would be to interpret the assertion and understand what needs to be changed to correct the violation. This would force every user of Clean++ to be proficient in CCEL.

An error message is an interpreted string which may contain references to bound metavariables and to the builtin variables `ConstraintId`, `ConstraintFile`, and `ConstraintLine` which print the unique identifier, file, and line that the constraint violated is defined at. Variables are denoted by { ... }. For instance, when using NIHCL[4], a programmer would want require that every class derived from "`Object`" declare the `isA` function with the constraint:

```
// The member function Object::isA must be redefined in all
// subclasses of class Object
RedefineisA(
  Class B | (B.name() == "Object");
  Class D | (D.is_descendant(B));
  MemberFunction B::f1 | (f1.name() == "isA");

  Assert(D.redefines(f1));
} "{D.file()}(line {D.begin_line()}) constraint {ConstraintId} in
\"{ConstraintFile}\": \n
Class {D.name()} does not define function isA.";
```

If class `MySubclass` does not declare the `isA` function, the violation message reported is:

```
mysubclass.h(Line 5) constraint RedefineisA in "constraints.ccel" :
    Class MySubclass does not define function isA.
```

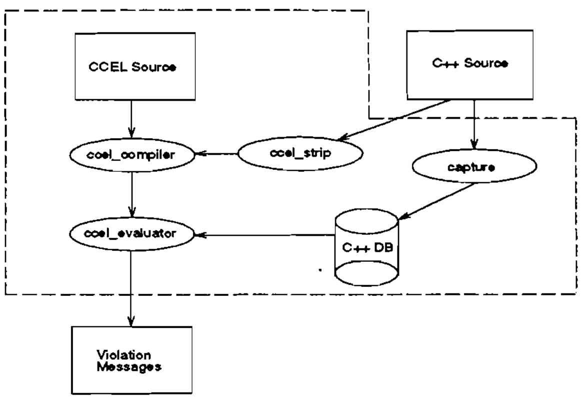

Figure 2: Clean++ Architecture

If the programmer does not define an error message for the constraint, a standard error message showing the value of the metavariables that caused the assertion to be violated and the identifier, line number and file of the constraint violated is reported. The default error message for the previous example is :

```
constraint RedefineisA in "constraints.ccel" (line 36):
    Class B = Object on line 20 of file "Object.h"
    Class D = MySubclass on line 5 of file "MySubclass.h"
    MemberFunction f1 = isA on line 23 of file "Object.h"
```

## 4 Prototype Architecture

There are two possible approaches to implementing Clean++. The first approach takes the constraints and generates a custom program that reads the C++ source and checks for violations of the constraints. The second approach extracts data from the C++ program and stores it in a database. The constraints are then converted into queries over the database. If a constraint is violated, its corresponding query will have a non-null result containing information about the violations.

The first approach requires generation of a custom constraint-checking program for each set of constraints, but for the second approach, a single constraint-interpreter suffices. As a result, we chose the second approach, because only one program is needed to apply multiple sets of constraints to the same system.

Figure 2 shows the components of Clean++. Clean++ constraints can be specified in one or more files or within a C++ program by embedding them in C++ comments. (This is similar to the way lint error messages can be controlled from within C source.) All features of CCEL can be used inside the C++ source, but we expect programmers will use it primarily to specify constraints specific to a class or file within the C++ source, i.e. to associate a constraint with the C++ source to which it applies. By associating constraints with C++ source, developers who want to derive a new class from a base class or create an instance of a class can see any constraints that apply to a class. For example, a constraint stating that all subclasses of a class must redefine a particular member function would be best put in the C++ source file for the class so that programmers know that they will need to define that member function. A more generic constraint, such as every class name must begin with an upper case letter, might go in a file containing style constraints.

The Clean++ architecture is made up of the following independent components: ccel_strip, ccel_compiler, ccel_evaluator, and capture. The ccel_strip program is a simple preprocessor that searches the C++ source files and extracts any constraints embedded in comments. The constraints from the C++ source and any constraint files are passed to ccel_compiler, which uses the C preprocessor to process any macros or file inclusions. ccel_compiler then translates the constraints into queries over the C++ database. The ccel_evaluator program executes the queries and outputs any violations of the constraints.

The capture program parses the C++ files and places their semantic structure in the C++ repository. There are several existing systems that can capture the structure and semantics of C++ programs. Such systems include REPRISE[14], CIA++[5], and XREFDB[7]. Of these, REPRISE seems most suited for use with CCEL, and our prototype implementation uses REPRISE for the database portion of Clean++.

For comparison purposes, the dotted lines in Figure 2 enclose the functionality of the lint tool for C programs. As is clear from the diagram, Clean++ gives the programmer more control by allowing the programmer to access the inner architecture of the checker and customize which constraints are checked. In particular, CCEL users can modify the CCEL source (i.e., the set of constraints to be enforced), while users of lint are unable to modify the conditions detected by lint.

## 5 Related Work

Support for formal design constraints in the form of assertions or annotations was designed into Eiffel [10], has been grafted onto Ada in the language Anna [9], and has been proposed for C++ in the form of A++ [2, 1]. This work, however, has grown out of the theory of abstract data types [8], 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). CCEL has a different focus. It has little concern for the semantics of functions;[1] however, it allows programmers to express constraints involving virtually any kind of declaration. As such, it is able 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., metaclass-specific naming conventions); this is also outside the purview of semantics-based constraint systems.

GENOA [3] is a language-independent application generator that can be used to generate a wide variety of code analysis tools. GENOA specifications consist of actions to be performed at nodes of an attributed parse tree. Unlike CCEL, which is specifically designed for C++ *programmers*, GENOA is designed for *compiler writers*. GENOA users must know the structure of the programming language parse tree, because applications based on GENOA are actually custom-designed traversals of this tree. CCEL hides this kind of grammatical detail, and is hence much easier to learn and use. The downside, of course, is that CCEL cannot be as expressive as applications taking full advantage of the generality of the GENOA approach.

[1] In fact, the current version of CCEL cannot express anything at all about function *definitions*. However, enhancing it so that it can is the next logical extension to the language.

## 6 Status

To date, our work on Clean++ has focused on developing a workable architecture for the system and on the design and implementation of CCEL. We have implemented a parser for CCEL and have completed the bulk of semantic analysis. Currently, we are concentrating on the implementation of a constraint evaluator for the internal representation of CCEL constraints; we are using REPRISE as the source of information about C++ source code. We expect to have a fully functional prototype for Clean++ before this research is presented at the USENIX C++ conference.

## References

[1] Marshall P. Cline and Doug Lea. The Behavior of C++ Classes. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 81–91, September 1990.

[2] Marshall P. Cline and Doug Lea. Using Annotated C++ . In *Proceedings of C++ at Work - '90*, pages 65–71, September 1990.

[3] Premkumar T. Devanbu. GENOA – a customizable, language- and front-end independent code analyzer. In *Proceedings of the International Conference on Software Engineering*, May 1992.

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

[5] Judith E. Grass and Yih-Farn Chen. The C++ Information Abstractor. In *USENIX C++ Conference Proceedings*, pages 265–277, 1990.

[6] Moises Lejter, Scott Meyers, and Steven P. Reiss. Adding Semantic Information To C++ Development Environments. In *Proceedings of C++ at Work-'90*, pages 103–108, September 1990.

[7] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for Maintaining Object-Oriented Programs. In *Proceedings of the Conference on Software Maintenance*, October 1991. This paper is largely drawn from two other papers [11, 6].

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

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

[10] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

[11] Scott Meyers. Working with Object-Oriented Programs: The View from the Trenches is Not Always Pretty. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 51–65, September 1990.

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

[13] Scott Meyers and Moises Lejter. Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++. In *USENIX C++ Conference Proceedings*, pages 29–40, April 1991.

[14] David S. Rosenblum and Alexander L. Wolf. Representing Semantically Analyzed C++ Code with Reprise. In *USENIX C++ Conference Proceedings*, pages 119 – 134, April 1991.

## A Additional Examples

The constraints that follow supplement the examples given in the body of the extended abstract. They serve to help demonstrate not only the expressiveness of the CCEL language itself, but also the kinds of constraints that C++ programmers might well want to enforce. [2]

```
// Subclasses must not redefine an inherited non-virtual member function.
NoNonVirtualRedefines (
    Class B, D;
    MemberFunction B::M;

    if (D.is_descendant(B) && D.redefines(M))
        Assert(M.is_virtual());
) "Class {B.name()} redefines an inherited non-virtual member function.";

// The return type of the assignment operator must be a reference to the class
ReturnTypeOfAssignmentOp (
    Class C1;
    MemberFunction C1::m1;

    if (m1.name() == "operator=")
        Assert(((m1.is_reference()) && (m1.type() == C1)));
) "The assignment operator for class {C1.name()} does not return a reference
to class {C1.name()}.";

// If a class contains a pointer member, the copy constructor must be defined.
CopyConstructorDefined (
    Class C;
    MemberVariable C::v1;

    if (v1.is_pointer())
       Assert(
           [ MemberFunction C::f1;
             Parameter      f1(p1); |
                 ((f1.name() == C.name()) && (f1.num_params() == 1) &&
```

[2] Although the grammar does not allow newlines in error message text, the error messages were separated for presentation purposes.

```
                    (p1.type() == C) && (p1.is_reference())))]);
) "A copy constructor should be defined for class (C.name()) because it contains
 the pointer member {v1.name()}";

// Members should be declared in the order public, protected, private
MemberDeclOrdering (
Class  C;

Assert(!([Member C::pub_mem | (pub_mem.is_public());
           Member C::non_pub_mem | (!non_pub_mem.is_public()); |
             (pub_mem.begin_line() > non_pub_mem.begin_line())] ||

         [Member C::prot_mem | (prot_mem.is_protected());
           Member C::priv_mem | (priv_mem.is_private()); |
             (prot_mem.begin_line() > priv_mem.begin_line())] ||

         [Member C::priv_mem | (priv_mem.is_private());
           Member C::non_priv_mem | (!non_priv_mem.is_private()); |
             (non_priv_mem.begin_line() > priv_mem.begin_line())]));

) "Class {C.name()) has members that are not declared in the order
        public, protected, private.";

// Derived classes should not redefine an inherited default parameter
// of a virtual function.
NoRedefineOfDefaults (
    Class          B, D;
    MemberFunction D::df, B::bf;
    Parameter      df(p1), bf(p2);

    if ((D.is_descendant(B)) && (df.is_virtual()) &&
        (df.redefines(bf)))
         Assert(((p1.position() == p2.position()) &&
          (p1.is_default() == p2.is_default())) ||
          (p2.position() != p1.position()));
) "Member function {D.name()}::{df.name()} redefines member function
{B.name()}::{bf.name()} but parameters {p1.name()} of function {df.name()} and
{p2.name()} of function {bf.name()} have different defaults.";


// Multiple inheritance hierarchies should not be diamond-shaped.
HierarchyStructure (
    Assert(![ Class A, B1, B2, C; |
    ((B1.is_descendant(A)) && (B2.is_descendant(A)) &&
     (C.is_descendant(B1) &&
     (C.is_descendant(B2)) && (B1.name() != B2.name()))))]);
) "Class {A.name()} is a ancestor of (B1.name()} and {B2.name()} which are
ancestors of {C.name()}." ;
```

## B CCEL Grammar

What follows is the YACC grammar for the CCEL prototype; semantic actions have been excluded.

```
%union {
    char          *cval;
    int            ival;
}

%token  <cval> STRING IDENT CONSTRAINT_CLASS_KEY  CLASS_KEY FILE_KEY
%token  <cval> C_PLUS_PLUS_OBJECT_KEY FUNCTDEF_KEY TYPEDOBJECT_KEY TYPE_KEY
%token  <cval> PARAMETER_KEY MEMBERVARIABLE_KEY MEMBERFUNCTION_KEY ASSERT_KEY
%token  <cval> ENABLE_KEY DISABLE_KEY IF_KEY MEMBER_KEY VARIABLE_KEY INTEGER
%token  <cval> FUNCTION_KEY ARRAY_DIM_KEY NAMEDOBJECT_KEY INDIRECTION_KEY
%token  <cval> CLASSORSTRUCT_KEY STRUCT_KEY
%type   <cval> var_name
%%

constraint_file : constraint_file constraint_group  |
                  constraint_group ;


constraint_group : constraint_class   |
                   constraint_list ;

constraint_class : CONSTRAINT_CLASS_KEY constraint_class_ident '{'
                   constraint_list '}' ';';

constraint_class_ident : unique_id ;

constraint_list : constraint_list constraint |
                  constraint

unique_id : IDENT ;

constraint : opt_constraint_scope constraint_ident '('
             constraint_body ';' '}'
             opt_message ';'

constraint_body : variable_decls constraint_condition |
                  select_constraint;

constraint_ident : unique_id ;

opt_constraint_scope : scope_obj obj_name ':' |
                       ;

scope_obj : CLASS_KEY    | FILE_KEY     | FUNCTION_KEY | VARIABLE_KEY ;

obj_name : filename   |
           IDENT ;

filename : STRING ;

opt_message : STRING |
              ;

variable_decls : variable_decls variable_decl |
                 ;
```

```
variable_decl : type var_name_list optional_cond ';'
      ;

type : C_PLUS_PLUS_OBJECT_KEY |
       FUNCTDEF_KEY           |
       TYPEDOBJECT_KEY        |
       TYPE_KEY               |
       VARIABLE_KEY           |
       MEMBER_KEY             |
       FUNCTION_KEY           |
       CLASS_KEY              |
       PARAMETER_KEY          |
       MEMBERVARIABLE_KEY     |
       MEMBERFUNCTION_KEY     |
       ARRAY_DIM_KEY          |
       NAMEDOBJECT_KEY        |
       INDIRECTION_KEY        |
       CLASSORSTRUCT_KEY      |
       STRUCT_KEY;

var_name_list : var_name_list ',' var_name |
                var_name ;

var_name : IDENT ':' ':' IDENT  |
           IDENT '(' IDENT ')'  |
           IDENT '[' IDENT ']'  |
           IDENT;

optional_cond : '|' expression |
                ;

constraint_condition : IF_KEY '(' expression ')' ASSERT_KEY '(' expression ')' |
                       ASSERT_KEY '(' expression ')'                          |
                       ;

param_list : expression_list |
             ;

expression_list : expression_list ',' expression |
                  expression;

expression : simple_expression             |
             simple_expression '=' '=' simple_expression |
             simple_expression '!' '=' simple_expression |
             simple_expression '<'     simple_expression |
             simple_expression '>'     simple_expression |
             simple_expression '<' '=' simple_expression |
             simple_expression '>' '=' simple_expression ;

simple_expression :
     term |
     simple_expression '|' '|' term;

term : factor |
       term '&' '&' factor;
```

```
factor : STRING   |
         INTEGER |
         IDENT function_list   |
         '!' factor |
         '(' expression ')' |
         '[' variable_decls '|' expression ']' ;

function_list : function_list '.' IDENT '(' param_list ')' |
          ;

select_constraint : on_or_off constraint_selector ;

on_or_off : ENABLE_KEY |
            DISABLE_KEY ;

constraint_selector : selected_constraint |
                      constraint_class_ident ;

selected_constraint : constraint_class_ident ':'':' constraint_ident;
```

## C CCEL Tokens

What follows is the LEX source for the CCEL prototype.

```
ws       [ \n\t]
letter   [A-Za-z]
digit    [0-9]
integer  {digit}+
punt     [_]
%%

{ws}                               {skip_whitespace();}
{integer}                          {sscanf(yytext, "%d", yylval.ival);
                                    return(INTEGER);}
"//"                               {skip_comment();}
"ArrayDim"                         {return(ARRAY_DIM_KEY);}
"ConstraintClass"                  {return(CONSTRAINT_CLASS_KEY);}
"Class"                            {return(CLASS_KEY);}
"File"                             {return(FILE_KEY);}
"Function"                         {return(FUNCTION_KEY);}
"Variable"                         {return(VARIABLE_KEY);}
"C++Object"                        {return(C_PLUS_PLUS_OBJECT_KEY);}
"FunctBody"                        {return(FUNCTBODY_KEY);}
"TypedObject"                      {return(TYPEDOBJECT_KEY);}
"Type"                             {return(TYPE_KEY);}
"Member"                           {return(MEMBER_KEY);}
"Parameter"                        {return(PARAMETER_KEY);}
"MemberVariable"                   {return(MEMBERVARIABLE_KEY);}
"MemberFunction"                   {return(MEMBERFUNCTION_KEY);}
"NamedObject"                      {return(NAMEDOBJECT_KEY);}
"Indirection"                      {return(INDIRECTION_KEY);}
"ClassOrStruct"                    {return(CLASSORSTRUCT_KEY);}
```

```
"Struct"                            {return(STRUCT_KEY);}
"Assert"                            {return(ASSERT_KEY);}
"if"                                {return(IF_KEY);}
"enable"                            {return(ENABLE_KEY);}
"disable"                           {return(DISABLE_KEY);}
{letter}({letter}|{digit}|{punt})*
                                    {yylval.cval =
                                          new char[strlen(yytext)+1];
                                     strcpy(yylval.cval, yytext);
                                     return(IDENT);}
\".*\"                              {yylval.cval =
                                          new char[strlen(yytext)+1];
                                     strcpy(yylval.cval, yytext);
                                     return(STRING);}
.                                   {return(yytext[0]);}
```