BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M6

"CCEL: The C++ Constraint Expression Language"

by

Yueh-hong Lin

This research project by Yueh-hong Lin 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.

Date 5/20/93

Steven P. Reiss

CCEL: The C++ Constraint Expression Language

An Annotated Reference Manual

Version 0.5

Yueh-hong Lin Scott Meyers

May 18, 1993

Contents

1 Introduction **1**

2 Overview of CCEL **1**

3 CCEL Lexical Conventions **4**
3.1 Tokens 4
3.2 Comments 6
3.3 Identifiers 6
3.4 Keywords 6
3.5 Literals 7

4 CCEL Classes **7**

5 CCEL Variables **9**

6 CCEL Expressions **11**

7 CCEL Constraints **13**
7.1 Constraints 13
7.2 Constraint Qualifiers 17
7.3 Constraint Classes 20

8 Violation Messages **23**

9 CCEL Classes and Member Functions **25**
9.1 Int 25
9.2 String 26
9.3 C++Object 27
9.4 NamedObject 27
9.5 Type 28
9.6 Class 32
9.7 Template 33
9.8 TypedObject 33
9.9 Function 36
9.10 Variable 37
9.11 AnyParameter 37
9.12 Parameter 37
9.13 TypeParameter 38
9.14 Member 38
9.15 MemberFunction 39
9.16 DataMember 39
9.17 TypeMember 39

10 Unsupported Features in CCEL 40

11 Request for Comments 40

A CCEL Grammar 41

A.1 Programs 41

A.2 Constraint Classes 41

A.3 Constraints 41

A.4 Constraint Qualifiers 42

A.5 Expressions 43

A.6 Variables Declarations 44

A.7 Names 44

1 Introduction

C++ is an expressive language, but it does not allow software developers to say many of 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++:

- *The member function M in class C must be overridden in all classes derived from C.* This is an example of a **design constraint**, because the constraint is specific to a particular class, C, and a particular member function in that class, M. This kind of constraint is common in general-purpose class libraries. For example, the NIH class library [3] contains many functions which must always be redefined if the library is to function correctly.

- *If a class declares a pointer data member, it must also declare an assignment operator and a copy constructor.* This is an example of a design-independent **implementation constraint**. Failure to adhere to this constraint almost always leads to incorrect program behavior [5].

- *All class names must begin with an upper case letter.* This is an example of one of the most common kinds of **stylistic constraints**. Most software development teams adopt some type of naming convention for identifiers, violations of which 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 different sets of constraints. Our approach to this problem is the development of a new language, CCEL ("Cecil")—the C++ Constraint Expression Language, that allows software developers to specify a wide variety of constraints and to have a system automatically detect violations of those constraints.

CCEL-I, which is the version of CCEL described in this document, supports the expression of constraints on C++ *declarations*; it has no vocabulary for specifying constraints on C++ *definitions*. A future CCEL-II will add to CCEL-I the ability to specify constraints on definitions.

Our work on this document and on the CCEL language itself is an ongoing endeavor, and we are quite interested in your reactions to both the language and this specification of it. For information on how to send comments to us, see Section 11.

2 Overview of CCEL

A CCEL program consists of a set of CCEL *constraints*, *constraint qualifiers*, and *constraint classes*, which are to be imposed on C++ sources. Here, we say *C++ sources* instead of *C++ programs* because they do not need to contain the function `main`. They could be arbitrary sets of C++ source files or C++ libraries. The C++ sources to be checked by a CCEL program are the *target C++ sources.*

CCEL constraints are used to describe the rules about C++ sources. They are loosely based on expressions in the predicate calculus, allowing programmers to make assertions (modeled on the `assert` macro) involving existentially or universally quantified CCEL variables. If an assertion fails, violation messages would be reported. The following is an example of a CCEL constraint:

```
// Every base class must have a virtual destructor:
File "ListNode.H" : VirtualDtorInBase (
   Class B;                          // for each class B
   Class D | D.is_descendant( B );   // for each class D derived from B

   Assert( MemberFunction B::bmf; |  // there must exist a member
                                     // function bmf in B such that:
           bmf.name() == "~" + B.name() && bmf.is_virtual() );
);
```

In CCEL, the characters // start a comment to the end of the line, just as in C++. An English translation for this constraint is:

> For all classes B and D declared in the file ListNode.H 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. "ListNode.H" specifies where in the target C++ sources VirtualDtorInBase applies. The file ListNode.H is called the *applicable scope* of VirtualDtorInBase. An applicable scope can be the entirety of the target C++ sources or any combination of the files, the functions, and the classes in the target C++ sources. Class and MemberFunction are CCEL classes. CCEL classes are the type system of CCEL (see Figure 1). B, D, and bmf are CCEL variables. Variables B and D are of type Class, so their values may range over the classes in the target C++ sources. bmf is of type MemberFunction and its values may range over the member functions. Because B and D are declared outside the Assert clause, they are universally quantified variables. On the other hand, bmf is an existential quantified variable ,because it is declared inside the Assert clause.

The functions is_descendant(), name(), is_virtual(), operator==(), operator+(), and operator&&() are CCEL class member functions (see Table 1). 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 literal. The Assert clause comprises the essence of the constraint. It asserts that for all possible bindings of the universally quantified variables B and D, there must exist at least one binding of the existentially quantified variable bmf such that the expression inside it,

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

evaluates to true.

If this constraint is violated, a violation message is to be reported. For example, suppose the target C++ sources are the following:

```
class Object {
   public:
   virtual char *is_a();
};

class Ball : public Object {
   ...
};
```

Classes `Object` and `Ball` violate the constraint `VirtualDtorInBase`, because `Object` is the base class of `Ball`, but it does not have a virtual destructor. A message about this violation is reported as follows:

```
"constraint.ccel", line 28: VirtualDtorInBase violated:
        B = Object ("objects.H", line 15)
        D = Ball ("objects.H", line 20)
```

The message says that the constraint `VirtualDtorInBase` beginning on line 28 in the CCEL source file `constraint.ccel` is violated because the variable `B` can be bound to the C++ class `Object`, which begins on line 15 in the C++ source file `object.H`, and because variable `D` can be bound to class `Ball`, which begins on line 20 in the file `object.H`. The above violation message is in the default format. However, formats of violation messages may be defined by CCEL programmers themselves.

Constraint qualifiers are used to change the applicable scopes of constraints. For example, the constraint qualifier

```
File "TreeNode.H" : EnableVirtualDtorInBase (
   Enable VirtualDtorInBase;
);
```

makes `virtualDtorInBase` apply to the file `TreeNode.H` in addition to `ListNode.H`.

Constraint classes are used to group individual constraints and constraint qualifiers. For example, a constraint class can be used to group the constraints related to `VirtualDtorInBase` together:

```
// Some constraints about base classes:
File "ListNode.H" : RequirementsOnBase {
   Class B;                          // for each class B
   Class D | D.is_descendant( B );   // for each class D derived from B

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

   CtorInBase (
      Assert( MemberFunction B::bmf; |
              bmf.name() == B.name() );
   );
};
```

As shown above, the constraints in a constraint class may share the applicable scope and universally quantified variables.

Summarizing, the features of CCEL are:

- **CCEL classes** represent the components of C++ sources, such as classes, functions, variables, etc. They serve as the type system of CCEL. CCEL classes are described in Section 4.

- **CCEL variables** are declared with one of the CCEL classes as their types and are to be bound to the components represented by the CCEL class. CCEL variables are described in Section 5.

- **CCEL expressions** are constructed with function calls to CCEL class member functions. They may be used to specify the restrictions on variable bindings, the goals of `Assert` clauses, or the formats of violation messages. CCEL expressions are described in Section 6.

- **CCEL constraints** are used to express rules about C++ sources by making assertions involving existentially and/or universally quantified CCEL variables. CCEL constraints are described in Section 7.1.

- **Applicable scopes** are the part of target C++ sources to which a constraint applies. They may be specified in constraint declarations or may be changed by constraint qualifiers. Constraint qualifiers are described in Section 7.2.

- **Constraint classes** are used to group together individual constraints which may share the applicable scope and universally quantified variables. Constraint classes can also be used to group constraint qualifiers. Constraint classes are described in Section 7.3.

- **Violation messages** are reported if CCEL constraints are violated. Violation messages may be reported in a default format or in user-defined formats. Violation messages are described in Section 8.

In the rest of this document, the CCEL language is formally described. The syntax notation used in this document is similar to that used in the ARM [2]. The syntactic categories are indicated in *slanted* type, and literal words and characters in `typewriter` type. Alternatives are listed on separate lines. An optional symbol is indicated by the subscript *opt.* Design issues, examples, etc, which have no place in a reference manual, are presented as annotations as the ARM does.

3 CCEL Lexical Conventions

3.1 Tokens

There are five kinds of tokens in CCEL: identifiers, keywords, literals, operators, and other separators. In CCEL, the white space characters are treated in the same way as in C++.

CCEL Class	Member Function
AnyParameter	Int position()
C++Object	String file()
	Int begin_line()
	Int end_line()
Class	
DataMember	
Function	Int num_params()
	Int is_inline()
	Int is_friend(Class)
Int	Int operator == (Int)
	Int operator < (Int)
	Int operator ! ()
	Int operator && (Int)
	Int operator || (Int)
	Int operator != (Int)
	Int operator > (Int)
	Int operator <= (Int)
	Int operator >= (Int)
Member	Int is_private()
	Int is_protected()
	Int is_public()
MemberFunction	Int is_virtual()
	Int is_pure_virtual()
	Int overrides(MemberFunction)
NamedObject	String name()
	Int scope_is_global()
	Int scope_is_file()
Parameter	Int has_default_value()
String	Int operator == (String)
	Int operator < (String)
	Int match(String)
	String operator + (String)
	Int operator != (String)
	Int operator > (String)
	Int operator <= (String)
	Int operator >= (String)

CCEL Class	Member Function
Template	Int is_class_template()
	Int is_function_template()
Type	Int has_name(String)
	Type basic_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)
	Int operator != (Type)
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()
TypeMember	
TypeParameter	
Variable	Int scope_is_local()

Table 1: CCEL Class Member Functions. The functions shown in the upper box of each CCEL class are primary functions. The ones in the lower box are secondary functions. A primary function is necessary for the expressive power of CCEL. A secondary function is just a convenience function and is defined in terms of the primary functions.

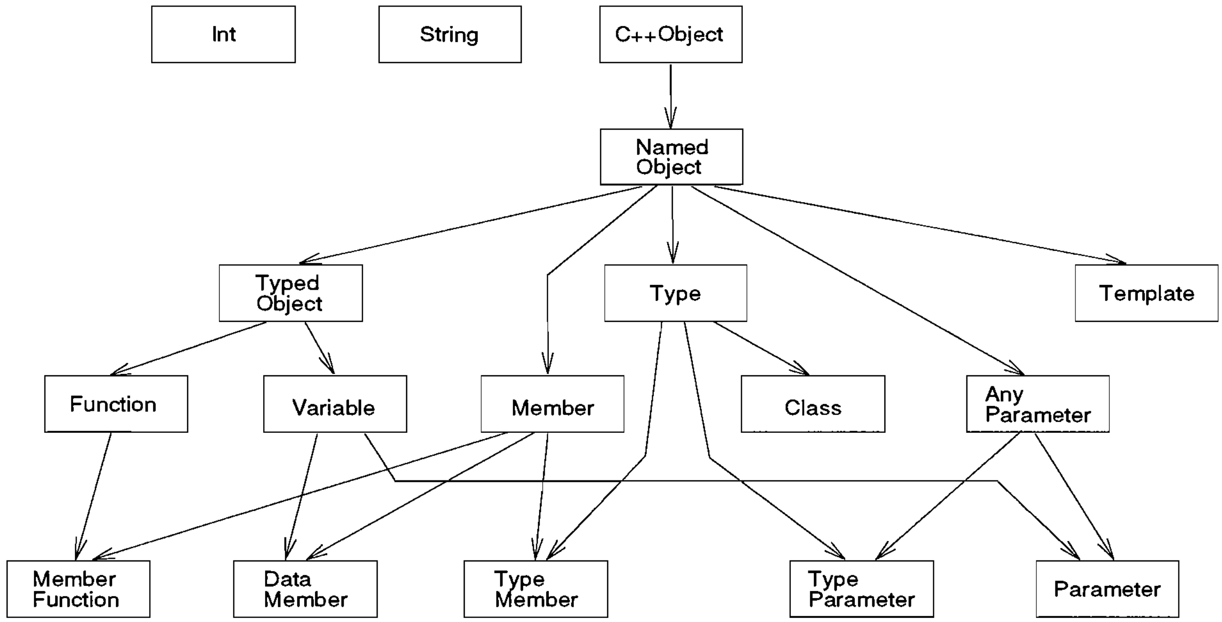

Figure 1: CCEL Class Hierarchy

3.2 Comments

The characters // start a comment in CCEL in exactly the same way as C++ . The C++ block comment /* ... */ is not supported in CCEL.

3.3 Identifiers

The rule for making a legal identifier in CCEL is the same as in C++.

3.4 Keywords

The following identifiers are reserved as keywords in CCEL and cannot be used in any other fashion:
Assert Enable Disable

- Unlike C++, the names of built-in types are not reserved as keywords in CCEL. Thus, as few restrictions as possible are imposed on CCEL programmers. There is no ambiguity in the CCEL grammar whether those names are reserved as keywords or not.

The ASCII representation of CCEL programs uses the following characters as operators or for punctuation:

```
! ( ) + { } | ; : " < > , .
```

and the following character combinations are used as operators:

```
<= >= == != && || ::
```

Each is a single token.

3.5 Literals

CCEL allows integer literals and string literals. An integer literal must be given in the decimal form and is treated in the same way as a decimal integer constant in C++. A string literal is given in the same way as a string literal in C++ except that those escape sequences which represent single quotes, question marks, octal numbers and hexadecimal numbers are not provided in CCEL. This is because single quotes and question marks are not punctuation characters in CCEL, and CCEL is not used for numerical purposes.

4 CCEL Classes

CCEL classes are the type system in CCEL. In C++, fundamental types such as int, char, etc are not treated as classes. Members and inheritance relationships are not allowed for them. Unlike C++, all CCEL types are classes and may have members and inheritance relationships. Based on the object-oriented model, CCEL classes are arranged in a multiple inheritance *is-a* hierarchy and represent the components of C++ sources, such as templates, types, classes, functions, variables, etc (see Figure 1). CCEL class member functions are defined to access information about the properties of the components (see Table 1).

The following is a list of CCEL classes:

- Int: The integer type.
- String: The string type.
- C++Object: The components of C++ sources, such as types, functions, etc.
- NamedObject: The named components of C++ sources, including templates, types, functions, variables.
- Type: C++ types.
- Class: C++ classes, i.e. class, struct, and union.

  ▪ Most C++ programmers might want to draw distinctions between class and struct. However, class, struct, and union are all called *classes* in C++ by Stroustrup's book, *The Annotated C++ Reference Manual* [2] (in the rest of this document, this book will be referred as *C++ ARM*). To be consistent with C++, we use one CCEL class, Class, to represent all *generally speaking* C++ classes, including struct, class, and union.

- Template: C++ templates, including class templates and function templates.
- TypedObject: The components of C++ sources that have types associated with them, including variables and functions.
- Function: C++ functions, including global functions, file static functions, and class member functions.

- **Variable**: C++ variables, including global variables, file static variables, function parameters, local variables, and class data members.

  ▪ The class Variable also represents C++ function parameters because function parameters are just like variables in C++ . Furthermore, most constraints generally imposed on global variables, local variables, and class data members are also intended to be imposed on function parameters.

- **AnyParameter**: C++ parameters, including function parameters and template parameters.
- **Parameter**: C++ formal function parameters or formal template non-type parameters.
- **TypeParameter**: The actual type parameters of C++ template instantiations.
- **Member**: C++ class members, including member functions, data members, and even type members (types declared in a class).
- **MemberFunction**: C++ class member functions.
- **DataMember**: C++ class data members.
- **TypeMember**: The C++ types which are declared or introduced by `typedef` inside a class.

Detailed descriptions of each CCEL class and its member functions are presented in Section 9.

▪ When designing CCEL, we determined which classes were needed by examining in detail the concepts important to C++ programmers and the constraints which programmers commonly want to express. Then, we classified the concepts we had identified into *CCEL classes*, such as the C++ classes and member functions, and into *properties* of CCEL classes (accessed by member functions), such as the protection level of a C++ member function. Finally, we determined the CCEL class hierarchy by analyzing the features of the CCEL classes and by moving the common features up the hierarchy into more general CCEL classes.

While abstracting the concepts of C++ into CCEL classes, we often had to decide if a concept was a new CCEL class or if it could be expressed as member functions of the existing CCEL classes. For example, the only difference between a C++ class and a C++ 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 use a single CCEL class to represent both class and struct with a boolean member function indicating whether it is a class. A second possibility would be to use two CCEL classes to represent class and struct separately, with their common functionality abstracted to a base CCEL class (this has been tried in an earlier version of CCEL [1]).

We also combined concepts into one CCEL class when the differences were trivial and the additional complexity of having a new CCEL class outweighed the increased functionality. For example, we think the distinction between functions in general (i.e., both global and member functions) and global functions in particular is not significant, so the current CCEL class hierarchy has no single CCEL class specifically devoted to global functions.

5 CCEL Variables

Each CCEL variable must be declared with a CCEL class as its type. CCEL variables are like the tuple variables in a database query language: during evaluation, they are bound to values from the program information database of the target C++ sources [6]. For example, a CCEL variable of type **Member** may be bound to each class member in the target C++ sources.

There are two kinds of CCEL variables—*the universally quantified variables* and *the existentially quantified variables*. The way they are quantified depends on where they are declared. This will be explained in Section 7.1, *Constraints*.

All CCEL variables must be declared before use. A variable declaration has the form:

variable_declaration:
    class_name variable_specifier_list ;

class_name:
    identifier

variable_specifier_list:
    variable_specifier
    variable_specifier_list , *variable_specifier*

A *class_name* is the name of one of the CCEL classes which can be used to declare CCEL variables, i.e. **NamedObject** or any CCEL classes derived from **NamedObject**.

A *variable_specifier* has the form:

variable_specifier:
    variable_scoper $condition_{opt}$

variable_scoper:
    variable_name
    class_variable_name :: *variable_name*
    function_variable_name (*variable_name*)
    template_variable_name < *variable_name* >

variable_name:
    identifier

class_variable_name:
    identifier

function_variable_name:
    identifier

template_variable_name:
    identifier

For the first alternative of a *variable_scoper*, the type of the declaration must be one of the following CCEL classes which can be used to declare variables directly: **NamedObject**, **Type**, **Class**, **Template**, **TypedObject**, **Function**, and **Variable**. For example,

```
Class C;
```

declares the CCEL variable `C` of type **Class**. The variable is to be bound to each class in the target C++ source.

For the second alternative of a *variable_scoper*, *class_variable_name* is the name of a CCEL variable which has been declared of type **Class**. The type used in this declaration must be **Member**, **MemberFunction**, **DataMember**, or **TypeMember**. For example,

```
Class C;
MemberFunction C::mfunc;
```

declares the CCEL variable `mfunc` of type **MemberFunction**. The variable is to be bound to each member function of the class to which the CCEL variable `C` is bound.

For the third alternative of a *variable_scoper*, *function_variable_name* must be the name of a CCEL variable which has been declared of type **Function** or **MemberFunction**. The type used in this declaration must be **Parameter**. For example,

```
Class C;
MemberFunction C::mfunc;
Parameter mfunc(p);
```

declares the CCEL variable **p** of type **Parameter**. The variable is to be bound to each parameter of the member function to which the CCEL variable `mfunc` is bound.

For the last alternative of a *variable_scoper*, *template_variable_name* must be the name of a variable which has been declared of type **Template**. The type used in this declaration must be **AnyParameter**, **Parameter**, or **TypeParameter**. For example,

```
Template T;
TypeParameter T<C>;
```

declares the variable `C` of type **TypeParameter**. The variable is to be bound to each type parameter of the template to which the CCEL variable `T` is bound.

If present, a *condition* is a vertical bar followed by a CCEL expression (see Section 6):

condition:
    | *expression*

The expression here is a *binding restriction expression.* It specifies the restriction on the values to be bound to the variable being declared. The vertical bar means *such that* (as in set theory). Only those components in the target C++ sources which make the expression evaluate to true can be bound to the variable. The result of the expression must be an integer, because a boolean value is required here (see Section 9.1 about type **Int** and boolean values). For example, the following are CCEL variable declarations:

```
Function f1 | f1.name() == "quick_sort",
         f2;
```

The function `name()` is a member function of the class **NamedObject**. It returns the name of the named object. The possible bindings of the variable `f1` are the functions whose names are `quick_sort`. On the other hand, `f2` may be bound to each function in the target C++ sources.

6 CCEL Expressions

In CCEL, expressions are used to specify the restrictions on variable bindings, the goals of `Assert` clauses (see Section 7.1), or the formats of violation messages (see Section 8). A CCEL expression is constructed with function calls to CCEL class member functions. A call to an operator member function must be written in infix form rather than function call form. The operator precedence of CCEL member functions is the same as in C++. Parentheses can be used to enforce the order of the computation of operators.

For example, suppose `x` is a CCEL variable of type NamedObject. The following is a CCEL expression:

```
x.name() == "List"
```

In the above, the member function `name()` of the variable `x` is called first and `name()` returns a string. Then the member function `operator==` of the returned string is called with the string literal `"List"` as the parameter. The above expression may not be written as

```
x.name().operator == ("List")    // error !
```

This restriction makes the syntax of CCEL simpler.

A CCEL expression has the form:

expression:
    logical_or_expression

logical_or_expression:
    logical_and_expression
    logical_or_expression `||` *logical_and_expression*

logical_and_expression:
    equality_expression
    logical_and_expression `&&` *equality_expression*

equality_expression:
    relational_expression
    equality_expression `==` *relational_expression*
    equality_expression `!=` *relational_expression*

relational_expression:
    additive_expression
    relational_expression `<` *additive_expression*
    relational_expression `>` *additive_expression*
    relational_expression `<=` *additive_expression*
    relational_expression `>=` *additive_expression*

additive_expression:
    unary_expression
    additive_expression `+` *unary_expression*

unary_expression:
    literal
    (*expression*)
    postfix_expression
    ! *unary_expression*

literal:
    string_literal
    integer_literal

postfix_expression:
    variable_name
    postfix_expression . *member_function_name* (*expression_list*$_{opt}$)

variable_name:
    identifier

member_function_name:
    identifier

expression_list:
    expression
    expression_list , *expression*

Notice that according to the grammar, the infix notation of operator member function calls is allowed for literals, but the dot notation of ordinary member function calls is not allowed for literals. For example, the expressions

```
42.is_short()                // syntax error !
"strcmp".matches("strcmp")   // syntax error !
```

are rejected by the CCEL grammar. However, the expressions

```
500 > x.end_line()   // ok ! (x is a NamedObject variable)
"There should be no global function: " + f.name()   // ok ! (f is a Function variable)
```

are allowed in CCEL.

- To prevent the CCEL grammar from accepting expressions like this,

  ```
  (42).is_short()   // recognized by grammar as: (expression).is_short()
  ```

  parentheses are not allowed with the dot notation of ordinary member function calls. That is, the expressions,

  ```
  (mf).is_virtual()                          // syntax error !
  (x.name() + y.name()).match("StackNode")   // syntax error !
  ```

  are rejected as syntax errors.

7 CCEL Constraints

7.1 Constraints

CCEL constraints are used to express the rules about C++ sources. They are loosely based on expressions in the predicate calculus, allowing programmers to make assertions (modeled on the `assert` macro) involving existentially or universally quantified CCEL variables. If an assertion fails, violation messages are reported.

A CCEL constraint has the form:

constraint:
    original_applicable_scope$_{opt}$ *constraint_name* (*constraint_body*) *violation_message*$_{opt}$;

constraint_name:
    identifier

A *constraint_name* is an identifier and serves as the name of the constraint being declared. No two constraints declared outside constraint classes may have the same name.

An *original_applicable_scope* is used to specify the applicable scope of the constraint being declared. The applicable scope specified here is the *original applicable scope* because the applicable scope may be changed by constraint qualifiers (this will be explained in Section 7.2). An *original_applicable_scope* has the form:

original_applicable_scope:
    scope_selector_list :

scope_selector_list:
    scope_selector
    scope_selector_list , *scope_selector*

scope_selector:
    `File` *C++_file_selector*
    `Class` *C++_class_selector*
    `Function` *C++_function_selector*

C++_file_selector:
    string_literal

C++_class_selector:
    string_literal

C++_function_selector:
    string_literal

The original applicable scope of a constraint is the union of the C++ files, classes, and functions listed in the *original_applicable_scope*. If an *original_applicable_scope* is not given, the original applicable scope of the constraint being declared is the entire target C++ sources by default. In an *original_applicable_scope*, a *C++_file_selector* has the same syntax and semantics as the UNIX `sh` wildcards [4]. Whether the file names given here should include paths or not is implementation-dependent. This is because some environment parameters such as the current directory would be involved if file names not including full paths are allowed. A *C++_class_selector* and a *C++_function_selector* have the same syntax and semantics as the regular expressions of UNIX command `grep` [4]. For example, the following is an original applicable scope specification:

```
File "my_*.H", Class "^Human", Function "^Animal::move$" : MyRequirement (
   ...
);
```

The original applicable scope of the constraint `myRequirement` is all the files whose names match the wildcard `my_*.H`, all the classes whose names match the regular expression `^Human`, and the member functions whose names are `Animal::move`. As shown in this example, to refer to a function or a class which is not global, the full name including the scope resolution operator `::` must be given.

- For CCEL constraint libraries, sometimes it would be more convenient to let the original applicable scopes of the library constraints be null. Then library users can use constraint qualifiers to enable the constraints needed for their C++ sources. To make an original applicable scope null, a non-existing file name (or a class, a function) or an empty file can be used as:

```
File "" : SomeSpecificRule (
   ...
);

File "empty.H" : AnotherSpecificRule (
   ...
);
```

The first approach is safer.

A *constraint_body* has the form:

constraint_body:
    variable_declaration_list$_{opt}$ *assertion*

A *variable_declaration_list* is a list of CCEL variable declarations:

variable_declaration_list:
    variable_declaration
    variable_declaration_list variable_declaration

The CCEL variables declared here are universally quantified.

An *assertion* is the **Assert** clause which comprises the essence of a constraint. An **Assert** clause has one of the forms:

assertion:
    `Assert` (*variable_declaration_list* | *expression*) ;
    `Assert` (*expression*) ;
    `Assert` (*variable_declaration_list*) ;

In contrast to the fact that the CCEL variables declared outside the **Assert** clause are universally quantified, the variables declared inside the **Assert** clause are existentially quantified. No two variables declared in a constraint may have the same name. In the second alternative of the **Assert** clause shown above, there are no existentially quantified variables declared.

The expression inside an **Assert** clause is used to specify the goal of the assertion. The result of the expression must be an integer because a boolean value is required (see Section 9.1 about type **Int** and boolean values). The existentially quantified variables declared in the **Assert** clause and the universally quantified variables declared in the constraint can be used in the expression. The third form of an **Assert** clause is equivalent to the first form with the boolean literal **TRUE** as the expression:

```
Assert ( variable_declaration_list | TRUE ) ;
```

A constraint asserts that for *each* combination of the bindings of the universally quantified variables, there must exist *at least one* combination of the bindings of the existentially quantified variables such that the assertion expression is true. The assertion fails in the condition that for *some* combination of the bindings of the universally quantified variables, there exists *no* possible combination of the bindings of the existentially quantified variables such that the assertion expression is true. For the cases in which there are no universally quantified variables, or no existentially quantified variables, or for the cases in which the variables are declared but there are no possible bindings for them, the conditions to make the assertion fail or not are:

- Under the circumstance where there are no universally quantified variables declared, the assertion fails if there exists no possible combination of the bindings of the existentially quantified variables such that the assertion expression is true.

- Under the circumstance where there are universally quantified variables declared but there is no possible combination of bindings for them, the assertion never fails.

- Under the circumstance where there are no existentially quantified variables declared, the assertion fails if any combination of the bindings of the universally quantified variables makes the assertion expression evaluate to false.

- Under the circumstance where there are existentially quantified variables declared but there is no possible combination of bindings for them, the assertion fails no matter what the assertion expression is.

A violation message is to be reported for each combination of the bindings of the universally quantified variables which makes the assertion fail. For a constraint without any universally quantified variables, a single violation message is to be reported if the assertion fails. The *violation_message* at the end of a constraint declaration is used to specify the format of violation messages. This is described in Section 8.

The simplest possible constraint consists of only the constraint identifier and the **Assert** clause. For example:

```
// The global inline function debug must be declared:
GlobalInlineDebugExist (
   // there must exist a function F such that ...
   Assert( Function F | F.name() == "debug" &&
                        F.scope_is_global() &&
                        F.is_inline(); );
);
```

This constraint, having the identifier `GlobalInlineDebugExist`, declares the existential quantified variable `F` of type `Function`. It asserts that there must exist a function which has the name `debug` and which is a global inline function.

> ▪ In fact, any binding restriction expressions of the existential quantified variables can be moved into the assertion expression combined with *AND* operators to make a new constraint which retains the same semantics. For example, the above constraint can also be written as:
>
> ```
> InlineDebugExist (
>    Assert( Function F; |   // the following is the assertion expression:
>            F.name() == "debug" &&
>            F.scope_is_global() &&
>            F.is_inline() );
> );
> ```
>
> However, to put binding restriction along with variable declarations is sometimes more straightforward and makes the CCEL code clearer. Furthermore, the binding restriction on an individual CCEL variable may help the efficiency of constraint evaluation.

The following constraint involves both universally quantified variables and existentially quantified variables:

```
// Base::draw must be overridden in all classes derived from Base:
RedefineBaseDraw (
   Class B | B.name() == "Base";      // for class B whose name is "Base"
   Class C | C.is_descendant( B );    // for each class C derived from B
   MemberFunction B::bmf | bmf.name() == "draw";
                                      // for member function bmf in B, which has
                                      // name "draw"


   Assert( MemberFunction C::cmf | cmf.overrides( bmf ); );
                                      // there must exist a member function cmf in C
                                      // such that cmf override bmf
);
```

It declares the universally quantified variables `B`, `C`, and `bmf`, and the existential quantified variable `cmf`. It asserts that there must exist a member function in any class derived from `Base` such that the member function overrides the member function `Base::draw`.

▪ CCEL programmers must be careful when making a decision to declare a variable universally quantified or existentially quantified. For the above constraint, no violation message will ever be reported if the function `Base::draw` is missing. On the other hand, suppose that the variable `bmf` is declared as an existentially quantified variable:

```
RedefineBaseDraw (
   Class B | B.name() == "Base";
   Class C | C.is_descendant( B );

   Assert( MemberFunction B::bmf | bmf.name() == "draw";
                                     // asserts that Base::draw must exist
           MemberFunction C::cmf | rdraw.overrides( bmf ); );
);
```

`RedefineBaseDraw` is violated if the member function `Base::draw` is not declared in the target C++ sources.

It is sometimes useful to write a constraint such that any possible bindings of the universally quantified variables make the assertion fail. For example:

```
// No member functions in struct:
NoMemberFunctionInStruct (
   Class S | S.is_struct();   // for each class S which is a struct
   MemberFunction S::smf;     // for each member function smf in S

   // the assertion should fail for any combination of structs
   // and member functions that satisfy the above relationships
   Assert( FALSE );

);
```

However, it is meaningless to write a constraint with the Assert clause like this:

```
Assert( TRUE );
```

This is because there is no way to violate such a constraint and no violation message could ever be reported.

7.2 Constraint Qualifiers

A constraint qualifier is used to enable or disable some constraints or some other constraint qualifiers in its applicable scope. It has the form:

constraint_qualifier:
    original_applicable_scope$_{opt}$ *constraint_qualifier_name* (*constraint_qualifier_body*) ;

constraint_qualifier_name:
    identifier

constraint_qualifier_body:

enable_disable_statement_list

enable_disable_statement_list:
enable_disable_statement enable_disable_statement_list$_{opt}$

enable_disable_statement:
enable_statement
disable_statement

enable_statement:
`Enable` *constraint_selector_list* ;

disable_statement:
`Disable` *constraint_selector_list* ;

constraint_selector_list:
constraint_selector
constraint_selector_list , *constraint_selector*

constraint_selector:
constraint_name
constraint_qualifier_name
constraint_class_name
constraint_class_name :: *constraint_name*
constraint_class_name :: *constraint_qualifier_name*

constraint_name:
identifier

constraint_qualifier_name:
identifier

constraint_class_name:
identifier

Constraint classes will be described in Section 7.3. Here, just regard both *constraint_class_name* and *constraint_class_name*::*constraint_name* as *constraint_name*.

The *original_applicable_scope* in a constraint qualifier declaration has the same syntax and semantics as in a constraint declaration. The *constraint_qualifier_name* is an identifier and serves as the name of the constraint qualifier. No two constraint qualifiers declared outside constraint classes may have the same name. A constraint and a constraint qualifier declared outside constraint classes may *not* have the same name.

An *enable_statement* is used to enable those constraints or constraint qualifiers whose names are listed in the statement in the applicable scope of the constraint qualifier. A *disable_statement* is used to disable those constraints or constraint qualifiers whose names are listed in the statement in the applicable scope of the constraint qualifier. For example:

```
File "my.H" : MyLimit (
```

```
  ...
);

File "new.H" : EnableMyLimit (
  Enable MyLimit;
);

Function "^my_sort$" : DisableMyLimit (
  Disable MyLimit;
);
```

The original applicable scope of the constraint `MyLimit` is the file `my.H`. Enabled by the constraint qualifier `EnableMyLimit` and disabled by another constraint qualifier `DisableMyLimit`, `MyLimit` applies to the two files `my.H` and `new.H` but not the function `my_sort`. If there are both **Enable** statements and **Disable** statements acting on a constraint, the applicable scope of the constraint is decided by computing the scope union operations indicated by the **Enable** statements earlier than the scope minus operations indicated by the **Disable** statements. That is: enlarge the applicable scope first, then shrink it. The order of the statements shown in the CCEL source code does not matter. For example, adding to the above the constraint qualifier

```
DisableMyLimitAtAll (
  Disable MyLimit;
  Enable MyLimit;
);
```

makes `MyLimit` apply to nothing because it is disabled in entire target C++ sources at last.

A constraint qualifier can be used to enable or disable another constraint qualifier. However, a constraint qualifier cannot enable or disable itself. It also cannot enable or disable any constraint qualifier which affects its applicable scope. To re-enable a disabled constraint, another constraint qualifier has to be used to disable the constraint qualifier which disables the constraint. For example, adding to the above one more constraint qualifier

```
Function "^new.H$" : DisableDisableMyLimitAtALL (
  Disable DisableMyLimitAtALL;
);
```

makes the constraint `MyLimit` apply to the file `new.H`, not including the function `my_sort`.

> ▪ It should be common that the persons who use CCEL constraints to check their C++ sources are not the persons who wrote the constraints. This is one of the most important inspirations for us to support the constraint qualifier mechanism in addition to the original applicable scope specification in CCEL. It is convenient to take a set of CCEL constraints and then use a set of constraint qualifiers to decide their applicable scopes without touching the code of these constraints.
>
> ▪ In contrast to a constraint class that can be used to group the constraints which have repetitive universally quantified variables, a constraint qualifier can be used to group the constraints which have no repetitive universally quantified variables. For example:

```
// Class names must begin with a capital letter:
File "" : ClassNameCapitalLeading (
   Class C;

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

// Function names must begin with a lower case letter:
File "" : FunctionNameLowerCaseLeading (
   Function f;

   Assert( f.name().match( "^[a-z]" ) );
);

// A set of naming rules:
File "" : NamingConventions (
   Enable ClassNameCapitalLeading, FunctionNameLowerCaseLeading;
);
```

Notice that the original applicable scopes of the above constraints are all null. To make both `ClassNameCapitalLeading` and `FunctionNameLowerCaseLeading` apply to some part of the target C++ sources, someone can just enable the constraint qualifier `NamingConventions` instead of enabling two individual constraints.

7.3 Constraint Classes

A constraint class is used to group individual constraints and constraint qualifiers. It has the form:

constraint_class:
  original_applicable_scope$_{opt}$ *constraint_class_name* { *constraint_class_body* } ;

constraint_class_name:
  identifier

constraint_class_body:
  variable_declaration_list$_{opt}$ *constraint_or_qualifier_list*

constraint_or_qualifier_list:
  constraint_or_qualifier *constraint_or_qualifier_list*$_{opt}$

constraint_or_qualifier:
  constraint
  constraint_qualifier

Notice that the extent of constraint classes is demarcated by brackets `{...}`, while individual constraints use parentheses as their delimiters. Also, constraint classes cannot be nested.

A *constraint_class_name* is an identifier and serves as the name of the constraint class. No two constraint classes may have the same name. No constraint declared outside constraint classes may have the same name as a constraint class.

As for an individual constraint, the *original_applicable_scope* in a constraint class declaration is used to specify the original applicable scope of the constraint class. An original applicable scope specification is not allowed for individual constraints or constraint qualifiers inside a constraint class. The original applicable scope of a constraint or a constraint qualifier inside a constraint class is the original applicable scope of the constraint class.

A *constraint_or_qualifier_list* is a list of CCEL constraint or constraint qualifier declarations. No two constraints or constraint qualifiers declared in a constraint class may have the same name.

The *variable_declaration_list* is a list of variable declarations. It declares the CCEL variables which belong to the constraint class. A variable so declared is a *constraint class variable.* No two constraint class variables of a constraint class may have the same name. No variable declared in the constraints of a constraint class may have the same name as any constraint class variable of the constraint class. A constraint class variable can be used as a universally quantified variable in any constraint of the constraint class. A constraint class variable affects only those constraints which directly or indirectly use the variable. *Directly use* means that the constraint class variable is referred to explicitly. *Indirectly use* means that some other constraint class variable whose binding restriction expression involves the variable is directly or indirectly used in the constraint. For example:

```
// Members must be declared in this order:  public -> protected -> private:
MemberDeclOrdering {
   Class  C;
   Member C::pub  | pub.is_public();
   Member C::prot | prot.is_protected();
   Member C::priv | priv.is_private();

   PublicBeforeProtected (
      Assert( pub.begin_line() < prot.begin_line() );
   );

   PublicBeforePrivate (
      Assert( pub.begin_line() < priv.begin_line() );
   );

   ProtectedBeforePrivate (
      Assert( prot.begin_line() < priv.begin_line() );
   );
};
```

In the above example, the variable `C` acts as if it has been declared as a universally quantified variable in all the three individual constraints. `pub` is just like it has been declared in both `PublicBeforeProtected` and `PublicBeforePrivate`. `prot` is just like it has been declared in both `PublicBeforeProtected` and `ProtectedBeforePrivate`. `priv` is just like it has been declared in both `PublicBeforePrivate` and `ProtectedBeforePrivate`.

▪ The rule to decide whether the binding restriction of a variable involves some other variable or not is based on the syntax but not on the logical meaning. For example:

```
OddConstrClass {
   Class A;
   Class B | A.name() == "joke";  // logically, B's bindings don't involve A

   ConstrX (
      Class C | C.is_descendant( B );
      ...
   );
};
```

In the above example, B is regarded as related to A and both A and B will affect ConstrX.

There are some constraints which can be used together to make C++ struct just like C struct. They can be grouped together into a constraint class called LimitedStruct:

```
// Make C++ struct just like C struct:
File "x.H", File "y.H" : StructLikeCStruct {
  Class S | S.is_struct();

  OnlyPublicMember (
    Member S::m;

    Assert( m.is_public() );
  );

  NoMemberFunction (
    MemberFunction S::mf;

    Assert( FALSE );
  );

  NoInheritance (
    Class C | S.is_descendant( C );

    Assert( FALSE );
  );
};
```

The applicable scope of the constraint class StructLikeCStruct is the files x.H and y.H, such that each of the individual constraints OnlyPublicMember, NoMemberFunction, and NoInheritance applies to x.H and y.H. A constraint qualifier can be used to enable or disable a constraint class or an individual constraint of a constraint class. For example, add the following constraint qualifiers to the above example:

```
File "z.H" : EnableLimitedStruct (
  Enable LimitedStruct;
);

File "x.H" : DisableNoInheritance (
```

```
  Disable LimitedStruct::NoInheritance;
);
```

Now `LimitedStruct::OnlyPublicMember` and `LimitedStruct::NoMemberFunction` apply to the file `z.H` in addition to the files `x.H` and `y.H`. Disabled by `DisableNoInheritance` in the file `x.H`, `Limited::NoInheritance` applies only to the files `y.H` and `z.H`. As shown above, an individual constraint of a constraint class can be referred to outside the constraint class with the form:

constraint_class_name :: *constraint_name*

- To take out repetitive universally quantified variables to be declared as constraint class variables does not only make the code cleaner but also enhances the efficiency of constraint evaluation because the individual constraints can share the bindings.

8 Violation Messages

Violation messages are to be reported if a constraint is violated. For each combination of the bindings of the universally quantified variables which makes the assertion fail, there is a violation message to be reported. For a constraint without any universally quantified variables, a single violation message is to be reported if the assertion fails.

In a constraint declaration

constraint:
original_applicable_scope$_{opt}$ *constraint_name* (*constraint_body*) *violation_message*$_{opt}$;

the *violation_message* is used to specify the violation message format of the constraint being declared. It has the form:

violation_message:
expression

It is a CCEL expression whose result is a string. This string is reported as the violation message. Some built-in special tokens can be used as strings in the expression. These strings provide the information about the constraint (see Table 2). If a CCEL variable, say `x`, is referred to alone as a string in this expression, it is taken as the following string which provides the information about the binding of `x`:

x.name() ("*x.file()*", `line` *x.begin_line()*)

(see the example later in this section). Ordinarily, the CCEL member functions `begin_line()` and `end_line()` return values of type Int. However, if they are called in a *violation_message*, they return strings which represent the integers which they ordinarily return.

If a *violation_message* is not given in a constraint declaration, the following default format is used for the violation messages of the constraint:

default_violation_message:
    `ConstraintInfo + "violated:"` *variable_binding_info_list*$_{opt}$

variable_binding_info_list:
    variable_binding_info variable_binding_info_list$_{opt}$

variable_binding_info:
    `+ "\n\t" +` *variable_heading* + *variable_name*

For each universally quantified variable of the constraint, there is a corresponding *variable_binding_info*. The *variable_heading* is a string literal. If the name of the variable is `bmf`, the corresponding *variable_heading* is `"bmf = "`. The *variable_name* is referred to alone as a string here, so that it is to be taken as the string which provides the information about the binding as described earlier in this section.

▪ The default format of violation messages is deliberately designed to be compatible with that of standard UNIX software development tools, e.g., compilers. The benefit of this design decision is that programs that already know how to parse output messages from other tools will also be able to work seamlessly with a CCEL-based constraint-checking system.

For example, suppose that the constraint

```
// Subclasses must never redefine an inherited non-virtual member function:
NoNonVirtualOverrides (
   Class B;
   Class D | D.is_descendant( B );
   MemberFunction B::bmf;
   MemberFunction D::dmf | dmf.overrides( bmf );

   Assert( bmf.is_virtual() );
)
ConstraintInfo + ": Non-virtual " + B.name() + "::" + bmf.name() + "overridden by \n\t"
+ dmf + " of class " + D.name + " !";
```

is violated in the situation that the non-virtual member function `getSize()` of class `Table` is overridden by the member function `getSize()` of the derived class `HashTable`. The following violation message is reported:

```
"rules.ccel", line 5: NoNonVirtualOverrides: Non-virtual Table::getSize overridden by
        getSize ("hash_table.H", line 123) of class HashTable !
```

If the user-defined format is not present, the violation message is reported in the default format as follows:

```
"rules.ccel", line 5: NoNonVirtualOverrides violated:
        B = Table ("table.H", line 12)
        D = HashTable ("hash_table.H", line 17)
        bmf = getSize ("table.C", line 53)
        dmf = getSize ("hash_table.C", line 123)
```

Special token	Corresponding string
`ConstraintId`	the name of the constraint
`ConstraintFile`	the name of the file containing the constraint
`ConstraintLine`	the beginning line number of the constraint
`ConstraintInfo`	"*ConstraintFile*", `line` *ConstraintLine*: *ConstraintId*

Table 2: Special tokens to be used in violation message Formats

9 CCEL Classes and Member Functions

In this section, each CCEL class and its member functions are clearly described. In the object-oriented model of CCEL, the class member functions provide the way to access the properties of CCEL classes and the calls to member functions are the operators in CCEL expressions. The member functions are divided into two groups: primary member functions and secondary member functions. Primary functions are necessary for the expressive power of CCEL. Secondary member functions are just convenience functions and are defined in terms of primary functions. In the following descriptions, the word *this* stands for the object whose member function is called.

9.1 Int

The class Int is like the fundamental type int of C++ in that it represents the integer type. As in C++, Int is also used for boolean values in CCEL. 0 is false, all non-zero values are regarded as true. Two integer constants are defined by the system: TRUE of value 1 and FALSE of value 0. The class Int cannot be used to declare CCEL variables. However, some member functions return values of type Int.

Primary member functions:

`Int operator == ( Int i )`

returns TRUE if this integer is equal to the parameter `i`, returns FALSE otherwise.

`Int operator < ( Int i )`

returns TRUE if this integer is less than the parameter `i`, returns FALSE otherwise.

`Int operator ! ()`

Boolean NOT: returns TRUE if this integer is zero, returns FALSE otherwise.

`Int operator && ( Int i )`

Boolean AND: returns TRUE if both this integer and the parameter `i` are non-zero, returns FALSE otherwise.

Secondary member functions:

`Int operator != ( Int )`

`Int1 != Int2` $\equiv$ `!(Int1 == Int2)`

`Int operator > ( Int )`
`Int1 > Int2` $\equiv$ `!((Int1 == Int2) || (Int1 < Int2))`

`Int operator <= ( Int )`
`Int1 <= Int2` $\equiv$ `(Int1 < Int2) || (Int1 == Int2)`

`Int operator >= ( Int )`
`Int1 >= Int2` $\equiv$ `!(Int1 < Int2)`

`Int operator || ( Int )`
`Int1 || Int2` $\equiv$ `!( !Int1 && !Int2 )`

- The member function **operator ||** is the boolean *or* operator.

9.2 String

The class String represents character strings in CCEL. As for Int, the String class cannot be used to declare CCEL variables, but some member functions may return values of type String.

Primary member functions:

`Int operator == ( String s )`
returns TRUE if the standard C library function `strcmp` returns zero with these two strings as parameters, returns FALSE otherwise.

`Int operator < ( String s )`
returns TRUE if the C standard library function `strcmp` returns a negative value with this string as the first parameter and the parameter `s` as the second parameter, returns FALSE otherwise.

`Int match( String s )`
returns TRUE if this string matches the regular expression specified by the parameter `s`, returns FALSE otherwise. The syntax and semantics for the regular expression and matches are the same as for the UNIX command `grep` [4].

`String operator + ( String s )`
returns a String which is the concatenation of this string and the parameter `s`. For example, the result of the expression

```
"String" + "::" + "operator=="
```

is `"String::operator=="`.

Secondary member functions:

```
Int operator != ( String )
S1 != S2  ≡  !(S1 == S2)
```

```
Int operator > ( String para )
S1 > S2  ≡  !(S1 == S2 || S1 < S2)
```

```
Int operator <= ( String )
S1 <= S2  ≡  (S1 < S2) || (S1 == S2)
```

```
Int operator >= ( String )
S1 >= S2  ≡  !(S1 < S2)
```

9.3 C++Object

The class C++Object represents the components of C++ sources. For example, templates, types, functions, variables, etc are C++ source components. C++Object may not be used to declare CCEL variables.

Primary member functions:

`String file()`
returns the full name, including the absolute path, of the file which contains this C++Object.

`Int begin_line()`
returns the line number of the first lexical token of this C++Object.

`Int end_line()`
returns the line number of the last lexical token of this C++Object.

If the definition of this C++Object exists, the above functions are used to locate the definition. If the definition does not exist, the above functions locate one of the declarations of this C++Object.

- To abstract C++Object out as a base class makes it easier to add new CCEL classes to represent any kind of C++ source components in later versions of CCEL.

9.4 NamedObject

The class NamedObject represents those components of C++ sources that have names associated with them. Templates, types, functions, and variables in C++ sources are named objects. On the other hand, an expression or a `for` statement in C++ sources is not a named object. NamedObject and all CCEL classes derived from it can be used to declare CCEL variables. Anonymous classes in C++ sources are also regarded as named objects with names as empty strings.

Primary member functions:

String name()

returns the name of this NamedObject. The returned name is always a local name but never a fully qualified name. For example, if this NamedObject is bound to the member function pop of the class stack, name() returns the string "pop" rather than the string "stack::pop". The name returned by name() contains no unnecessary white space characters. For example, if this NamedObject is bound to the type const char*, name() returns the string "const char*" but never the string "const char *".

Int scope_is_global()

returns TRUE if this NamedObject is global (with external linkage), returns FALSE otherwise.

Int scope_is_file()

returns TRUE if this NamedObject is declared as static in a file scope, returns FALSE otherwise.

9.5 Type

The class Type represents types in C++ sources, including fundamental types such as int, char, etc, and derived types, which might be enum, union, struct, class, char*, const int, etc. The types returned by TypedObject::type() could be arbitrary C++ types except function signatures (see Section 9.8). However, a CCEL variable of type Type may only be bound to C++ fundamental types, enums, classes, and the types introduced by typedef. For example, for the C++ source

```
static const int TRUE = 1;

enum {
   ALPHA,
   BETA,
   GAMMA
} RayType;

typedef char* String;

class List;

int strcmp( const String s1, const String s2);
```

a CCEL variable T of type Type is to be bound to each fundamental type, char*, RayType, and List. Notice that if T is bound to char*, T.name() returns "String" rather than "char*".

- CCEL can be used to express the constraints about pointers to class member functions (e.g., void (SomeClass::*fp)()). For example:

```
// No function may take a parameter that is a pointer to member function:
NoPointerToMemberFunctionAsParameter (
   Class C;
```

```
    Function f;
    Parameter f(p) | p.type().name().match( "(" + C.name() + "::*)" );

    Assert( FALSE );
);
```

Primary member functions:

Int has_name(String s)

returns TRUE if the parameter s is the name of this Type or one of the synonyms of this Type, returns FALSE otherwise. For example, if this Type is bound to long, has_name() returns TRUE if either "long" or "long int" is passed as parameter. For the C++ source

```
class Node;
typedef Node* NodePt;
```

if this Type is bound to Node*, has_name() returns TRUE if either "Node*" or "NodePt" is passed as parameter. The names or synonyms recognized by has_name() may contain unnecessary white space characters.

Type basic_type()

returns this Type without any type declarator or type specifier. For example, if this Type is bound to static const char*[], basic_type() returns char.

▪ A C++ type, say the class Green, is a different type from it coming with type declarators and/or type specifiers, e.g., Green*. Suppose Green is derived from the class Color. It is absolutely false to say Green* is derived from Color. The member function basic_type() helps to make distinction between them. For example:

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

   Assert( mf.is_reference() && mf.type().basic_type() == C );
);
```

In C++, the type SomeClass& is not equivalent to SomeClass.

Int operator == (Type t)

returns TRUE if the two types are equivalent in C++, returns FALSE otherwise. C++ takes name equivalence for types. Two types are equivalent if and only if their names are the same.

Int is_convertible_to(Type t)

returns TRUE if this Type can be *implicitly* converted to the parameter t in C++, returns FALSE otherwise.

Int is_enum()
returns TRUE if this Type is an **enum** type, returns FALSE otherwise.

The following member functions deal with the properties of C++ class types: **class**, **struct**, and **union**. They are defined in the class Type because the problem introduced by `TypedObject::type()` (see Section 9.8, the commentary below the function `type()`). They just return FALSE if this Type is not a class.

▪ Although all class-related member functions are defined in Type, the CCEL class Class is still derived out because class is the main concept of C++ and is also what C++ programmers most want to impose constraints on. To declare a CCEL variable to be bound to C++ classes, it will be more convenient to write the statement

```
Class C;
```

rather than the statement

```
Type C | C.is_class();
```

Int is_class()
returns TRUE if this Class is a **class**, returns FALSE otherwise.

Int is_struct()
returns TRUE if this Class is a **struct**, returns FALSE otherwise.

Int is_union()
returns TRUE if this Class is an **union**, returns FALSE otherwise.

Int is_friend(Class c)
returns TRUE if this Class is a friend class of the parameter `c`, returns FALSE otherwise.

Int is_child(Class c)
returns TRUE if this Class is immediately derived from the parameter `c`, returns FALSE otherwise. *Immediately derived* means that the parameter Class appears in the base class list of the definition of this Class.

▪ The member function `is_child()` helps to express those constraints about class multiple inheritance. For example:

```
// No multiple inheritance is allowed:
NoMultiInheritance (
   Class A;
   Class B | B.is_child( A );
   Class C | C.is_child( A ) && C != B;
```

```
   Class D | ( D.is_descendant( B ) || D == B ) &&
             ( D.is_descendant( C ) || D == C );

   Assert( FALSE );
);
```

To express class multiple inheritance relationships, `is_child()` is necessary. Even with the member function `is_descendant()`, there is no way in CCEL to express a constraint about the following simplified multiple inheritance without `is_child()`:

```
class A { ... };
class B : public A { ... };
class C : public A, public B { ... };
```

With only `is_descendant()`, one might write down a CCEL constraint like this:

```
TryToFindSimplifiedMultiInheritance (
   Class a;
   Class b | b.is_descendant( a );
   Class c | c.is_descendant( a ) && c.is_descendant( b );

   ...
);
```

However, this constraint is incorrect because the C++ classes in the inheritance chain

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
```

are also possible bindings for the CCEL variables `a`, `b`, and `c`.

`Int is_descendant( Class c )`
returns TRUE if this Class is derived from the parameter `c`, returns FALSE otherwise. The *is-descendant* relationship are just the transitive closure of the *is-child* relationship.

`Int is_virtual_descendant( Class c )`
returns TRUE if this Class is a virtual descendant of the parameter `c`, returns FALSE otherwise. In CCEL, a C++ inheritance path is called a *virtual inheritance path* if and only if the first edge of the path is specified as virtual inheritance. A C++ class `D` is called a *virtual descendant* of a C++ class `B` if and only if `D` is descendant of `B` and *all* inheritance paths from `B` to `D` are virtual inheritance paths. The fact that `D` is a virtual descendant of `B` ensures that there is only one subobject of `B` in an object of `D`.

- The member function `is_virtual_descendant()` helps to express the following constraint:

```
// Multi-inheritance only allowed with virtual inheritance:
DiamondOnlyWithVirtual (
   Class A;
   Class B | B.is_child( A );
   Class C | C.is_child( A ) && C != B;
   Class D | ( D.is_descendant( B ) || D == B ) &&
             ( D.is_descendant( C ) || D == C );

   Assert( D.is_virtual_descendant( A ) );
);
```

`Int is_public_descendant( Class c )`

returns TRUE if this Class is a public descendant of the parameter `c`, returns FALSE otherwise. In CCEL, a C++ inheritance path is called a *public inheritance path* if and only if *all* edges of the path are specified as public inheritance. A C++ class `D` is a *public descendant* of a C++ class `B` if and only if there exists *any* public inheritance path from `B` to `D`.

- The member function `is_public_descendant()` helps to express the following constraint:

```
// Multiple public inheritance only allowed for virtual inheritance:
MultiPubInheritanceMustBeVirtual (
   Class A;
   Class B | B.is_child( A ) && D.is_public_descendant( A );
   Class C | C.is_child( A ) && E.is_public_descendant( A ) && C != B;
   Class D | ( D.is_public_descendant( B ) || D == B ) &&
             ( D.is_public_descendant( C ) || D == C );

   Assert( B.is_virtual_descendant( A ) &&
           C.is_virtual_descendant( A ) );
);
```

Secondary member functions:

`Int operator != ( Type )`

`T1 != T2` $\equiv$ `!(T1 == T2)`

9.6 Class

The class Class represents class types in C++ sources, including **class**, **struct** and **union**. The member functions involving C++ classes are defined in the class Type because of the problem introduced by `TypedObject::type()` (see Section 9.8, the commentary below the function `type()`).

9.7 Template

The class Template represents templates in C++ sources, including class templates and function templates.

Primary member functions:

`Int is_class_template()`

returns TRUE if this Template is a class template, returns FALSE otherwise.

Secondary member functions:

`Int is_function_template()`

`T.is_function_template()` $\equiv$ `! T.is_class_template()`

9.8 TypedObject

The class TypedObject represents those components of C++ sources which have types associated with them. C++ variables and functions are typed objects. On the other hand, a C++ type or a C++ template is not a typed object. CCEL regards the type of a C++ function as the type of its return value although C++ regards the type of a function as its signature.

▪ CCEL regards the type of a C++ function as the type of its return value instead of its signature because this is simpler and more useful. For the constraints involving C++ function parameters, the CCEL class Parameter can be used.

Primary member functions:

`Type type()`

returns the type of this TypedObject. The type of a C++ typed object includes all type declarators and type specifiers which are used to declare this typed object (see Table 3). The name of the type of a typed object is the lexical string used to declare it. Thus, if a type synonym is used to declare a typed object, the name of the type of the type object is the type synonym. For example:

```
typedef char* String;
String s1;
char *s2;
```

If this TypedObject is bound to `s1`, `type().name()` returns `"String"`. If this TypedObject is bound to `s2`, `type().name()` returns `"char*"`. However, both `type().has_name("String")` and `type().has_name("char*")` return TRUE whether this TypedObject is bound to `s1` or `s2`.

▪ The member function `type()` introduces a problem coming with such a constraint:

```
// The type of the parameter of some function must be derived from some class
LimitedParameterType (
   Function f | ...;
   Parameter f( p ) | ...;
```

Binding of this TypedObject	Returned value of type()
`int i;`	`int`
`unsigned long i;`	`unsigned long int`
`const int *&i;`	`const int*&`
`static volatile double v[];`	`static volatile double[]`
`typedef char *String;` `String msg;`	`char*`

Table 3: Returned values of type()

```
    Assert( Class Base | ...; | f.type().is_descendant( Base ) );
);
```

Sometimes, it is necessary to call class-related member functions for the type of a typed object (`P.type().is_descendant( Base )` in the above example). This means that `is_descendant()` and other class-involved member have to be functions defined in CCEL class Type because the return type of `TypedObject::type()` is Type.

Another solution without defining those member functions in Type is to introduce auxiliary CCEL variables. The above constraint can be rewritten as following:

```
LimitedParameterType (
    ...
    Assert( Class Base | ...;
            Class C | C == p.type(); |
            C.is_descendant( Base ); );
);
```

However, this solution is not adopted because to introduce auxiliary CCEL variables is inconvenient and may cause overhead in constraint evaluation.

<u>`Int num_indirections()`</u>
returns the number of indirection levels of this TypedObject. The number of indirections levels of a typed object is the number of pointer indirect accesses (specified with *) (see Table 4).

<u>`Int is_reference()`</u>
returns TRUE if this TypedObject is a reference (specified with &), returns FALSE otherwise.

<u>`Int is_static()`</u>
returns TRUE if this TypedObject is static, returns FALSE otherwise.

<u>`Int is_volatile()`</u>
returns TRUE if this TypedObject is volatile, returns FALSE otherwise.

Binding of this **TypedObject**	Number of indirection levels
`int i;`	0
`char **s;`	2
`int box[10][10][10]`	3 (since `int [][][]` is compatible with `int ***`)
`int *link_heads[];`	2
`typedef char *String;` `String *table;`	2 (`String*` implies `char**` since a `typedef` simply declares a synonym of a type but not really introduce a new type.)
`struct Node { Node *next; };` `Node head;`	0 (A `struct` itself is a type and head is of type `Node`.)
`int &a;`	0
`int *&a;`	1 (`num_indirections()` counts ONLY pointer indirect accesses (specified with `*`) but not reference (specified with `&`).

Table 4: Number of indirection levels of **TypedObject**

`Int is_const()`
returns TRUE if this **TypedObject** is a constant, returns FALSE otherwise. For example, `is_const()` returns TRUE if this **TypedObject** is bound to `char *const s`. However, it returns FALSE if this **TypedObject** is bound to a pointer-to-constant, e.g., `const char *s`. There is no facility in the current version of CCEL to check if a typed object is a pointer-to-constant.

`Int is_array()`
returns TRUE if this **TypedObject** is an array (specified with `[]`), returns FALSE otherwise (see Table 5).

`Int is_long()`
returns TRUE if the type of this **TypedObject** is a C++ long type, returns FALSE otherwise. Even for the systems on which the type **int** takes the same size as the type **long int** or the type **double** takes the same size as the type **long double**, **int** and **double** are not regarded as **long** types in CCEL.

`Int is_short()`
returns TRUE if the type of this **TypedObject** is a C++ **short** type, returns FALSE otherwise. Even for the systems on which the type **int** takes the same size as the type **short int**, **int** is not regarded as a **short** type in CCEL.

C++ source: `float final_grade( char* name, float grades[])`	
Binding of this TypedObject	Returned value of `is_array()`
`final_grade`	FALSE
`name`	FALSE
`grades`	TRUE

Table 5: Returned values of `is_array()`

`Int is_signed()`
returns TRUE if the type of this TypedObject is explicitly or implicitly specified as signed, returns FALSE otherwise.

`Int is_unsigned()`
returns TRUE if the type of this TypedObject is specified as unsigned, returns FALSE otherwise.

Secondary member functions:

`Int is_pointer()`
`is_pointer()` $\equiv$ `num_indirections() > 0`

9.9 Function

The class Function represents the functions in C++ sources, including global functions, file static functions, and class member functions.

Primary member functions:

`Int num_params()`
returns the number of formal parameters of this Function. The unspecified number of parameters (specified with ...) is not counted by `num_params()`. There is no facility in the current version of CCEL to check if a C++ function has an unspecified number of parameters.

`Int is_inline()`
returns TRUE if this Function is explicitly or implicitly declared as an inline function, returns FALSE otherwise.

▪ In C++, a member function with its body defined inside the class declaration is automatically taken as an inline function even without the keyword inline.

`Int is_friend( Class c )`
returns TRUE if this Function is a friend function of the parameter c, returns FALSE otherwise.

9.10 Variable

The class **Variable** represents variables in C++ sources, including global variables, file static variables, function parameters, local variables, and class data members.

Primary member functions:

`Int scope_is_local`

returns TRUE if this Variable is a local variable or a function parameter, returns FALSE otherwise.

- There is no member function `scope_is_class()`, in contrast with `scope_is_global()` and `scope_is_class()`, since it is equivalent to

```
!(scope_is_global() || scope_is_local())
```

Furthermore, the CCEL class DataMember is dedicated to C++ class data members. The declaration

```
Variable v | v.scope_is_class();
```

could be written as

```
Class C;
DataMember C::v;
```

9.11 AnyParameter

The class AnyParameter represents parameters in C++ sources, including function parameters and template parameters.

Primary member functions:

`Int position()`

returns the position of this AnyParameter in the parameter list. The position is numbered starting from zero, going from left to right.

9.12 Parameter

The class Parameter represents *formal* function parameters and non-type template parameters in C++ sources.

- The class Parameter can be used to express the following constraint:

```
// Templates in C++ sources can have only type parameters:
OnlyTypeParameterInTemplate (
   Template T;
   Parameter T< p >;     // p is to be bound to non-type parameters

   Assert( FALSE );
);
```

Primary member functions:

Int has_default_value()

returns TRUE if this Parameter is specified with a default value in either the declaration or the definition of the function, returns FALSE otherwise.

9.13 TypeParameter

The class TypeParameter represents the *actual* type parameters of template instantiations in C++ sources. There are no member functions defined in TypeParameter.

- TypeParameter represents the actual parameters of template instantiations instead of the formal parameters in template declarations because a template instantiation declares a new type and most C++ programmers are likely to impose constraints on the instantiations. For example, the following constraint imposes that the classes passed as the parameters to instantiate the template List must be derived from the class Node:

```
// Only the classes derived from Node can be passed as parameter to
// instantiate the template List:
OnlyNodeBasedClassForList (
   Template T | T.name() == "List";
   TypeParameter T< C >;

    Assert( Class B | B.name() == "Node"; |
            C.is_descendant( B ) );
);
```

9.14 Member

The class Member represents class members in C++ sources, including member functions, data members, and type members. In CCEL, a function (or a variable, or a type), m, is regarded as a member of C++ class C if and only if m is declared inside C. Thus, it is not possible for m to be a member inherited by C from some other class.

Primary member functions:

Int is_private()

returns TRUE if this Member is a private member, returns FALSE otherwise.

Int is_protected()

returns TRUE if this Member is a protected member, returns FALSE otherwise.

Secondary member functions:

Int is_public()

is_public $\equiv$!(is_private || is_protected)

9.15 MemberFunction

The class MemberFunction represents class member functions in C++ sources.

Primary member functions:

Int is_virtual()

returns TRUE if this MemberFunction is explicitly declared as a virtual member function or it overrides a virtual function, returns FALSE otherwise.

Int is_pure_virtual()

returns TRUE if this MemberFunction is a pure virtual member function, returns FALSE otherwise.

Int overrides(MemberFunction mf)

returns TRUE if this MemberFunction overrides the parameter `mf` in the class inheritance, returns FALSE otherwise. `overrides()` always return FALSE if `mf` is not a virtual function.

9.16 DataMember

The class DataMember represents class data members in C++ sources. There are no member functions defined in DataMember.

9.17 TypeMember

The class TypeMember represents the types which are declared or introduced by `typedef` inside a class in C++ sources. There are no member functions defined in TypeMember.

▪ Many C++ programmers like to have SmallTalk-like *Super* class to refer to the class which a class is immediately derived from (this is only for single inheritance). This is useful to incrementally refine inherited virtual member functions. Michael Tiemann suggested to use `typedef` in C++ to declare a synonym of the parent class as `Super`:

```
class Parent {
   virtual void foo();
   ...
};

class Child : public Parent {
private:
   typedef Parent Super;

   void foo()
   {
      Super::foo();
      ...
   }
   ...
```

```
};
```

The CCEL class TypeMember helps to express the following constraint to enforce this:

```
// "Super" refers to the parent class:
SuperAsParentClass (
   Class C;
   Class D | D.is_child( C );

   Assert( TypeMember D::tm | tm.name() == "Super" &&
                              tm == C; );
);
```

10 Unsupported Features in CCEL

The following are unsupported features in CCEL:

- CCEL has no concept of preprocessor macros.
- There is no facility in the current version of CCEL to check if a typed object is a pointer-to-constant, e.g., `const char*`.
- There is no facility in the current version of CCEL to check if a C++ function has an unspecified number of parameters (denoted as ...).

11 Request for Comments

Our work on this document and on the CCEL language itself is an ongoing endeavor, and we are quite interested in your reactions to both the language and this specification of it. Please send your comments to ccel@cs.brown.edu, or, if electronic mail is inconvenient or unavailable to you, to:

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

A CCEL Grammar

A.1 Programs

ccel_program:
  constraint_group_list

constraint_group_list:
  constraint_group constraint_group_list$_{opt}$

constraint_group:
  constraint_or_qualifier
  constraint_class

constraint_or_qualifier:
  constraint
  constraint_qualifier

A.2 Constraint Classes

constraint_class:
  original_applicable_scope$_{opt}$ *constraint_class_name* { *constraint_class_body* } ;

constraint_class_body:
  variable_declaration_list$_{opt}$ *constraint_or_qualifier_list*

constraint_or_qualifier_list:
  constraint_or_qualifier constraint_or_qualifier_list$_{opt}$

A.3 Constraints

constraint:
  original_applicable_scope$_{opt}$ *constraint_name* (*constraint_body*) *violation_message*$_{opt}$;

original_applicable_scope:
  scope_selector_list :

scope_selector_list:
  scope_selector
  scope_selector_list , *scope_selector*

scope_selector:
  `File` *C++_file_selector*
  `Class` *C++_class_selector*
  `Function` *C++_function_selector*

C++_file_selector:

string_literal

C++_class_selector:
string_literal

C++_function_selector:
string_literal

constraint_body:
variable_declaration_list$_{opt}$ assertion

assertion:
Assert (*variable_declaration_list* | *expression*) ;
Assert (*expression*) ;
Assert (*variable_declaration_list*) ;

violation_message:
expression

A.4 Constraint Qualifiers

constraint_qualifier:
original_applicable_scope$_{opt}$ constraint_qualifier_name (*constraint_qualifier_body*) ;

constraint_qualifier_body:
enable_disable_statement_list

enable_disable_statement_list:
enable_disable_statement enable_disable_statement_list$_{opt}$

enable_disable_statement:
enable_statement
disable_statement

enable_statement:
Enable *constraint_selector_list* ;

disable_statement:
Disable *constraint_selector_list* ;

constraint_selector_list:
constraint_selector
constraint_selector_list , *constraint_selector*

constraint_selector:
constraint_name
constraint_qualifier_name
constraint_class_name

constraint_class_name :: *constraint_name*
constraint_class_name :: *constraint_qualifier_name*

A.5 Expressions

expression_list:
expression
expression_list , *expression*

expression:
logical_or_expression

logical_or_expression:
logical_and_expression
logical_or_expression || *logical_and_expression*

logical_and_expression:
equality_expression
logical_and_expression && *equality_expression*

equality_expression:
relational_expression
equality_expression == *relational_expression*
equality_expression != *relational_expression*

relational_expression:
additive_expression
relational_expression < *additive_expression*
relational_expression > *additive_expression*
relational_expression <= *additive_expression*
relational_expression >= *additive_expression*

additive_expression:
unary_expression
additive_expression + *unary_expression*

unary_expression:
literal
(*expression*)
postfix_expression
! *unary_expression*

literal:
string_literal
integer_literal

postfix_expression:
variable_name

postfix_expression . member_function_name (expression_list$_{opt}$)

A.6 Variables Declarations

variable_declaration_list:
variable_declaration
variable_declaration_list variable_declaration

variable_declaration:
class_name variable_specifier_list ;

variable_specifier_list:
variable_specifier
variable_specifier_list , variable_specifier

variable_specifier:
variable_scoper condition$_{opt}$

variable_scoper:
variable_name
class_variable_name : : variable_name
function_variable_name (variable_name)
template_variable_name < variable_name >

condition:
| expression

A.7 Names

class_name:
identifier

member_function_name:
identifier

variable_name:
identifier

class_variable_name:
identifier

function_variable_name:
identifier

template_variable_name:
identifier

constraint_name:

identifier

constraint_qualifier_name:
identifier

constraint_class_name:
identifier

References

[1] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A Metalanguage for C++. In *USENIX C++ Conference Proceedings*, August 1992. Also available as Brown University Computer Science Department Technical Report CS-92-51, October 1992.

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

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

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

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

[6] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the Structure and Style of Object-Oriented Programs. In *Proceedings of the First Workshop on Principles and Practice of Constraint Programming (PPCP93)*, April 1993. Also available as Brown University Computer Science Department Technical Report CS-93-12, April 1993.