To appear in Reflection'96 Conference, April 21–24, 1996

Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –

Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda,
Jórg Nolte, Hiroshi Tezuka, Hiroki Konaka,
Munenori Maeda, Kazuto Kubota
Real World Computing Partnership
ishikawa@rwcp.or.jp

Abstract

We developed a compile-time metalevel architecture in C++, called the MPC++ metalevel architecture, to not only extend and modify language semantics, but also extend syntax. This architecture overcomes the imperative languages' issue of compile-time metalevel processing.

The proposed metalevel architecture has been implemented and tested. A typical metalevel programming example is given to demonstrate MPC++ metalevel architecture functionalities. Execution speeds of metalevel programs are measured to show the overhead resulting from metalevel processing in the current implementation.
Keywords:
compile-time metalevel architecture, C++, implementation

1 Introduction

Several metalevel architectures have been proposed to make systems extendable and modifiable. For example, a metalevel architecture of an object oriented programming language is implemented in such a way that the semantics of object creation, message passing or method invocation, and instance variable access are part of the language itself and reification and reflection mechanisms[3] are provided. A realistic application of such a programming system is to monitor objects for debugging[12] or to support the construction of an operating system[8, 14]. Some metalevel architectures can specify computational resources such as processors and communication networks in the distributed environment to achieve runtime optimization. These architectures enable the users to set scheduling strategy and load balancing[9, 10]. These systems focus on runtime behavior but do not focus on new language notations or syntax extensions.

An imperative language such as C and C++ has language constructs that support structural programming. In such a language, language semantics as well as syntax extensions are required to provide new language constructs to adapt the language to particular application domains such as parallel/distributed environments and real-time system. The MPC++ metalevel architecture was designed and implemented to meet this requirement.

The MPC++ metalevel architecture realizes compile-time metalevel processing. It defines an abstract compiler described in C++ and a modifi-

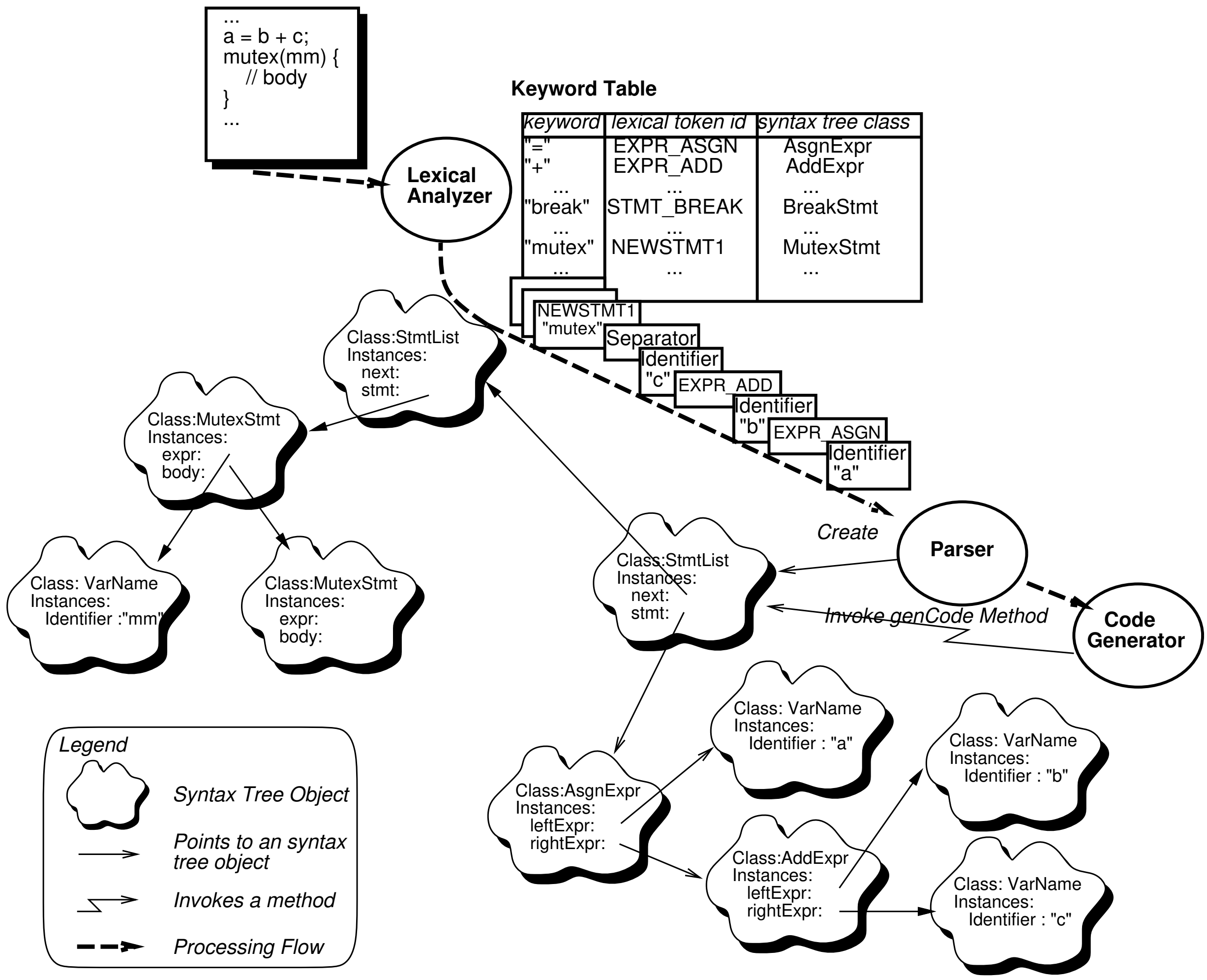

Figure 1: Abstract Compiler

cation facility called the MPC++ metaobject protocol. Programmers can easily introduce new syntax notations and redefine default code generation using the MPC++ metaobject protocol. Language extensions, debugging facilities, and machine dependent optimizations can be provided as metalevel source libraries. When the user wants to use some of the features provided by these libraries, she/he just includes those files that suit purposes with her/his source program.

This paper is organized as follows: Section 2 presents the design of the MPC++ metalevel architecture. After giving an overview of the architecture, issues and MPC++ answers on metalevel program reusability and handling nested statements in an imperative language will be discussed. In section 3, a realistic application of the MPC++ metalevel architecture, an introduction of the new statement `mutex`, will be shown to demonstrate the architecture's capability.

The current MPC++ implementation is briefly described in section 4. The MPC++ compiler system consists of the frontend and backend processors; frontend processor generates intermediate code while the backend processor generates target machine code. The frontend processor can interpret C++ code so that it performs MPC++ metalevel processing. Section 5 evaluates the current compiler system with respect to C++ interpreter and metalevel processing performance. Traditional approaches to support language extensions and related works on metalevel architectures are compared and discussed in section 6. Our conclusions are found in section 7.

2 Design

2.1 Overview

2.1.1 Abstract Compiler

The MPC++ metalevel architecture realizes compile-time metalevel processing. The architecture defines an abstract compiler consisting of a lexical analyzer, parser, and code generator as shown in Figure 1. The lexical analyzer takes a C++ program and produces a sequence of tokens. The parser receives lexical tokens and creates syntax tree objects to represent the parse tree. Then, the code generator invokes the `genCode` method of the syntax tree object. The `genCode` method of each syntax tree object is responsible for generating code. The default code generation is the C++ source code itself.

The metalevel object protocol has been designed to modify and extend C++ syntax and semantics. In section 2.2, the protocol on the lexical analyzer is designed so that the user can introduce new syntax. Section 2.3 focuses on the protocol on syntax tree objects which enables semantics modification and metalevel objects reusability.

2.1.2 Metalevel Definition

The keyword `$meta` is introduced to tell the compiler to shift to the metalevel. For example, the following example declares the integer variable `gi1` at the metalevel:

```
$meta int    gi1;
```

When the compiler meets:

```
$meta {
  printf("hello metaworld.\n");
}
```

The following message will appear at compile time, since the primitive function `printf` will be interpreted at the metalevel:

```
  hello metaworld.
```

```
$meta class ExprNode:public SynObj {
... };
$meta class AsgnExpr:public ExprNode {
private:
    ExprNode    *left;
    ExprNode    *right;
public:
    int         code(OStream);
    SynObj      *type();
};
```

Figure 2: Example of An Expression Syntax Tree Class

2.1.3 Syntax Tree Classes

A syntax tree class represents C++ syntax, such as expressions, statements, and declarations. For example, an assignment expression is represented by an instance of class `AsgnExpr` as defined in Figure 2. All syntax tree classes representing C++ expressions, called *expression syntax tree classes*, are derived from class `ExprNode` whose base class, known as a super class in Smalltalk terminology, is class `SynObj`. An expression syntax tree class has `code` and `type` methods that are responsible for generating code and telling type information, respectively.

The `code` takes an output stream, where code is generated, as its argument. During code generation, variable or type declarations are generated into the file scope, function scope, or structure scope. Declarations in the file scope are global while declarations in the function scope are local. Some declarations must appear in a structure as structure/class member data or types.

The MPC++ metalevel architecture defines four kinds of output streams; those names are `GlobalOS`, `LocalOS`, and `MemberOS` for the file, function, and structure scopes, respectively. Moreover, `CurrentOS` is provided to keep the currently generated place.

Another example of a syntax tree class is a *statement syntax tree* class, which represents a C++ statement. Figure 3 shows the `BreakStmt` statement syntax tree class, which represents the `break` statement. A statement syntax tree class is derived from class `StmtNode` whose base class is class `SynObj`. A statement syntax tree class has `code`

Table 1: Part of Extension Patterns

A token identifier, terminal symbol, is represented by a capital letter or special character surrounded by two 's while a nonterminal symbol is represented by an italic letter. The grammar shown above is not the actual rule for understandability.

Extension	Grammar
New Type Modifier	*TypeModifier* : `NEWSPECMODIFIER`
New Declarator Modifier	*DeclModifier* : `NEWDECLMODIFIER`
New Expressions	*Expr* : `NEWEXPR_UNARY`
	Expr : *Expr* `NEWEXPR_POSTFIX`
	Expr : *Expr* `NEWEXPR_INFIX` *Expr*
	Expr : `NEWEXPR_PREFIX` *Expr*
New Statements	*Stmt* : `NEWSTMT1` '(' *Expr* ')' *CompoudStmt*
	Stmt : `NEWSTMT2` '(' *Expr* ';' *Expr* ')' *CompoudStmt*
New Structural Declaration	*StructKey* : `NEWSTRCUTDECL`

```
$meta class StmtNode:public SynObj {
... };
$meta class BreakStmt:public StmtNode {
public:
    int         code(OStream);
};
```

Figure 3: Example of A Statement Syntax Tree Class

method to generate code.

Further details on syntax tree classes protocol are presented in Section 2.3.

2.2 Lexical Analyzer Protocol

We do not intend to allow users to extend any C++ syntax because good extensions must conform to some pattern. Moreover, the system would be too complicated if we allowed any syntax extensions the user wanted. Instead, the MPC++ metalevel architecture is designed so that an arbitrary grammar rule is not dynamically defined but extension patterns are predefined. The extension patterns are designed based on existing extended C++ language features as summarized in [5]. Among others, they include i) new modifiers for a specifier or a declarator, ii) new expressions, iii) new statements, and iv) new structural declarations as shown in Table 1. The parser contains the grammar for both C++ syntax and the extension patterns.

The lexical analyzer keyword table stores triple information, keywords, their lexical token values, and syntax tree classes. The primitive function `syntaxBind` was introduced to maintain the keyword table. For example, after the metalevel statement shown in Figure 4 is evaluated, triple information, a new keyword `mutex`, token value `NEWSTMT1`, and the syntax tree class called `MutexStmt`, are registered in the keyword table.

When the lexical analyzer sees the `mutex` statement shown in Figure 1, it can produce sequences such as:

```
NEWSTMT1, '(', Identifier, ...
```

When the `NEWSTMT1` token is produced for the `mutex` keyword, an instance of `MutexStmt` is also created. Then, the following extension pattern grammar is applied and the syntax tree object shown in Figure 1 is created:

```
"NEWSTMT1 '(' Expr ')' CompoundStmt"
```

It should be noted that the users may register other keywords for `NEWSTMT1` without syntax ambiguity. The lexical analyzer produces `NEWSTMT1`s for these registered keywords but creates different syntax tree objects according to the lexical analyzer keyword table.

```
$meta class MutexStmt : public NewStmt1 {
public:
    int         code(OStream);
     ... };
$meta {
  syntaxBind("mutex", NEWSTMT1, typeid(MutexStmt));
}
```

Figure 4: Mutex Statement

```
$meta class ModifedMutexStmt : public MutexStmt {
public:
    int         code(OStream);
     ... };
$meta {
  syntaxBind("mutex", NEWSTMT1, typeid(ModifiedMutexStmt));
}
```

Figure 5: Class ModifiedMutexStmt

2.3 Syntax Tree Object Protocol

In this subsection, a reusability issue of syntax tree objects is presented, along with its MPC++ answer. Then, section 2.3.2 describes the issue of handling the syntax and semantics of nested statements in an imperative language is described as well as the MPC++ answer.

2.3.1 Code Generation Stack

A mechanism for modifying existing code generation must be provided to make a metalevel program reusable. One way to do this is to write a subclass of the syntax tree class that the user wants to modify. For example, suppose that the new syntax `mutex` has been introduced and the user wants to modify code generation. The user simply writes the subclass `ModifiedMutexStmt` of the `MutexStmt` class as shown in Figure 5.

It seems that the inheritance mechanism helps us with respect to reusability. However, its ability is restricted in the following sense. Suppose that the new statement `mutex` has been implemented by class `MutexStmt` in the "mutex.h" file and somebody implemented its enhanced class called `ModifiedMutexStmt` in the "emutex.h" file. We want to add some debug statements for the `mutex` statement. If we include "mutex.h", `MutexStmt` must be the base class of the class that we program while `ModifiedMutexStmt` muse be the base class of our program in "emutex.h". In this way, we need to know which class has been implemented for the desired syntax and have to implement a modified class for each target class.

Our solution to this reusability issue is to support a kind of delegation mechanism for syntax tree objects, called a *code generation stack protocol*, in addition to the inheritance mechanism. Every user-defined syntax tree object has a pointer, called the *code generation stack pointer*, that points to another syntax tree object created from the default syntax tree class or another user defined class.

The program shown below is the skeleton of the *code generation stack protocol*. When the `genCode` method is invoked by the parser, it calls the `code` method. If the return value is `GEN_CONT`, the `genCode` method calls the `genCode` method of the object pointed to by the code generation stack pointer (`cgsptr` in the example) which will call the `code` method of that object. This procedure continues until the default syntax tree object is reached or the code method returns the `GEN_END` value.

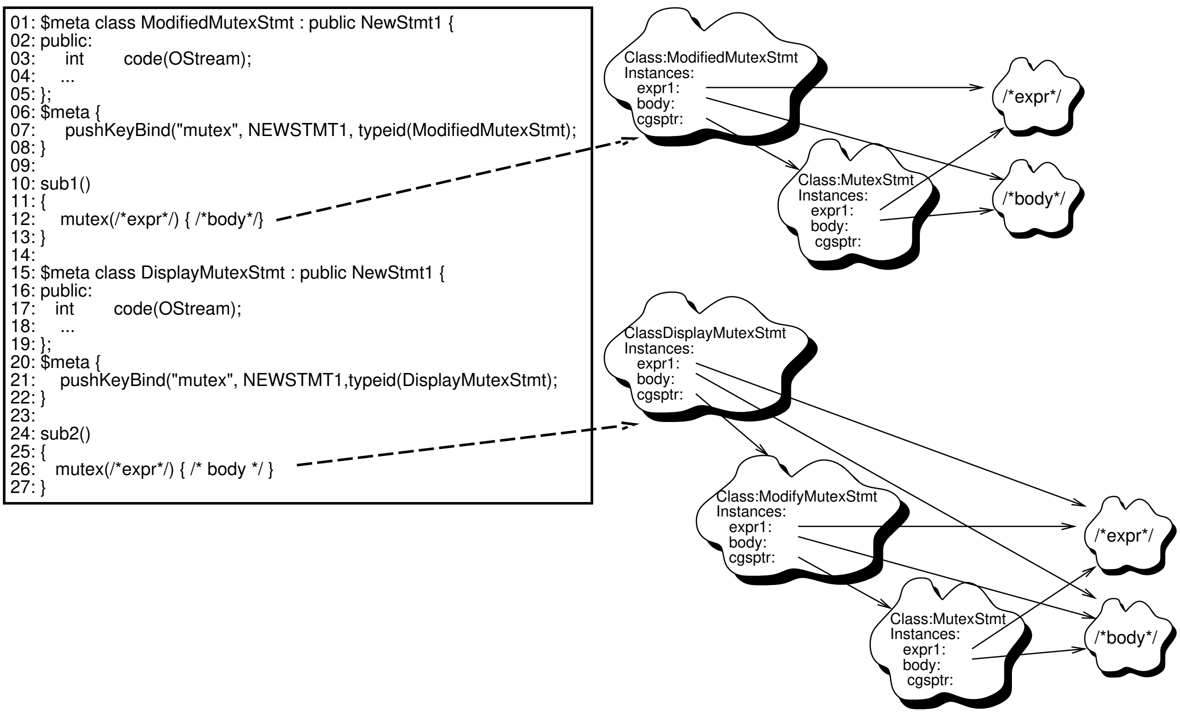

Figure 6: Code Generation Stack Protocol Example

```
SynObj::genCode(OStream os)
{
 if (this->code(os) == GEN_CONT
     && cgsptr) {
     cgsptr->genCode(os);
 }
}
```

The primitive functions `pushKeyBind` and `popKeyBind` are provided to support the *code generation stack protocol.* The function `pushKeyBind`, whose arguments are a lexical token and a class type identifier, tells the lexical analyzer that an instance of the specified class has been created and its code generation stack pointer is bound to an instance of the previous class of the lexical token when it produces the lexical token.

An example of the code generation stack protocol is shown in Figure 6. Suppose that the `mutex` statement has been implemented in class `MutexStmt`. When the compiler reaches the mutex statement in line 12, an instance of class `ModifiedMutexStmt` is created and its code generation stack pointer points to an instance created from `MutexStmt` since `ModifiedMutexStmt` has been registered by the `pushKeyBind` primitive in line 7.

When the compiler reaches the mutex statement in line 26, class `DisplayMutexStmt` is created and its code generation stack pointer points to an instance created from class `ModifiedMutexStmt`, whose code generation stack pointer points to the `MutexStmt` object since `DisplayMutexStmt` has been registered by the `pushKeyBind` primitive in line 21.

2.3.2 Context-dependent Processing

The code generation for some statements, such as `break`, requires context-dependent code generation. In the example given in Figure 7, the `break` statement in line 5 exits the loop beginning in line 3 while line 9 exits from the switch statement starting in line 6. Though parse node objects for both `break` statements are the same, the code generation must be different.

Instead of handling such context-dependent code generation in a node object, statement nodes

```
sub(int i, int j)
{
    while (i > 0) {
        ...;
        if (...) break;
        switch (j) {
        case 0:
            ...;
            break;
        default:
            ...;
        }
    }
}
```

Figure 7: An Example of Context Dependent Processing

are changed according to context within MPC++ metalevel architecture. For example, the syntax tree node for the `break` statement inside the `while` statement is an instance of `BreakWhileStmt` while the syntax tree node inside the `switch` statement is an instance of `BreakSwitchStmt`.

We designed the lexical analyzer to perform such context dependent processing. Any block statement, such as `while` statement, is derived from `BlockStmt` to keep an environment of nested statements.

As shown in Figure 8, the `BlockStmt` has the `outer` instance pointing to the outer block statement. The static variable `curEnv` points to the most inner nested block statement during the code generation phase. The `parseEnter` method is invoked by the lexical analyzer when it produces a subclass of `BlockStmt`. The `parseEnter` method sets up the link of a nested block statement and then invokes the `enter` method defined in the subclass of `BlockStmt`.

For example, when the lexical analyzer meets a `while` keyword, it creates a `WhileStmt` class derived from `BlockStmt` by default. The lexical analyzer invokes `parseEnter` method and eventually invokes the `WhileStmt enter` method that will change a syntax tree class of `break` statement to `BreakWhileStmt`.

The lexical analyzer invokes the `BlockStmt parseLeave` method when it finishes parsing the corresponding block statement. The `parseLeave` method calls the `leave` method defined in `BlockStmt`'s subclass and resets the nested statement link. In the case of parsing the `while` statement, the `WhileStmt leave` method is invoked to recover the lexical analyzer keyword table.

The users may program the `enter` and `leave` methods of a user-defined statement syntax tree class so that context dependent processing is applied. A context-dependent example of a user defined new statement is shown in Section 3.

2.3.3 Syntax Tree Constant

The user can write a code generation program that creates objects from syntax tree classes at the metalevel. For example, the following metalevel program generates code for the expression "`1 + 2 + 3`."

```
$meta {
 ExprNode *en = new AddExpr(new Constant(1),
                            new Constant(2));
 ExprNode *top = new AddExpr(en,
                            new Constant(3));
 top->genCode(GlobalOS);
}
```

Looking at this example, it is hard to write and understand what program is generated. A syntax tree constant is designed to allow the user to define a syntax tree object easily. As shown in the following example, a syntax tree constant for an expression is surrounded with `` `( `` and `)` while a syntax tree constant for statements is surrounded with `` `{ `` and `}`:

```
`( 1 + 2 + 3 )
`{ int i; i = 10; }
```

The user may use variables in a syntax tree constant as shown below. An identifier variable begins with a dollar sign followed by a number. A type variable begins with two dollar signs followed by a number. The number represents the parameter position. A syntax tree constant uses the method `genCode`, which takes as arguments an output stream and values for variables in the syntax tree constant.

If the examples in Figure 9 are evaluated, the following code is generated. During code generation, the type name `foo`, the integer variable `a`, and the floating array variable `f` are registered in the abstract compiler name table so that refer-

```
class BlockStmt : public StmtNode {
private:
    BlockStmt           *outer;
public:
    static BlockStmt    *curEnv;
    int                 parseEnter() { outer = curEnv; curEnv = this;
                                       return this->enter(); }
    int                 parseLeave() { int  ret;
                                       ret = this->leave();
                                       curEnv = outer; }
    BlockStmt           *getOuter { return outer; }
};
```

Figure 8: Class BlockStmt

```
$meta {
    `($1 + $2).genCode(GlobalOS, 10, 20);
    `{ struct $1 { int i; }; }.genCode(GlobalOS, gensym("foo"));
    `{ $$1 $2; }.genCode(GlobalOS, base_typeid(int), gensym("a"));
    `{ $$1 $2[$3]; }.genCode(GlobalOS, base_typeid(float), gensym("f"), 10);
}
```

Figure 9: Syntax Tree Constant Example

ences to these types and variables are correctly compiled.

```
10 + 20;
struct foo { int i; };
int a;
float f[10];
```

2.4 Introduction of New Syntax

The MPC++ architecture provides easy syntax extension mechanisms for the user. For example, the `$syntaxdef` statement is designed to introduce new syntax. The program in Figure 4 is the same as the following:

```
$syntaxdef $statement "mutex"
           MutexStmt : public NewStmt1 {
 public:
  code(OStream);
};
```

3 Example

We have several examples using the MPC++ metalevel architecture. The technical report[5], describing the old MPC++ metalevel architecture, shows programs implementing i) the atomic class which creates an instance object where execution threads are performed in a mutually exclusive way, ii) a global pointer that enables remote memory read and write operations, iii) a persistent object, and iv) the `mutex` statement to realize the mutual execution of statements. Here, the implementation of the `mutex` statement is presented.

Suppose that a library designer provides class `Mutex` to set up the critical region, which has `lock` and `unlock` methods. The critical region is programmed in such a way that method `lock` must be invoked before the critical region is entered, and method `unlock` must be invoked on exiting the critical region. Sample code is shown below.

```
Mutex   mm;
void sub()
{
    mm.lock();
    /* critical region here */
    mm.unlock();
}
```

One risk of the above program is that the critical region would remain unfinished if the programmer forgot to invoke method `unlock`. Thus, as shown below, the library designer might want to provide a new statement `mutex` to realize the critical region using metalevel programming.

```
$meta class MutexBreak : public BreakStmt {
public:
  int            code(OStream);
};

$syntaxdef $statement "mutex" MutexStmt : public NewStmt1 {
private:
  SynObj         *mutexp;
  Identifier     *exitLabel;
public:
  int            code(OStream);
  int            enter() {  pushKeyBind("break", STMT_BREAK, "MutexBreak"); }
  int            leave() {  popKeyBind("break", STMT_BREAK); }
  SynObj         *getMutex() { return mutexp; }
  Identifier     *getLabel() { return exitLabel; }
};
```

Figure 10: Mutex Statement

```
Mutex   mm;
void sub()
{
    mutex(mm) {
        /* critical region here */
    }
}
```

The `break` statement inside the `mutex` statement may exit the `mutex` statement. We have to modify the code generation of statements `break`, `return`, and `goto` as they appear within the `mutex` statement so that the `unlock` method is called on exiting the statement. In this paper, we show how the `break` statement used within the `mutex` statement works.

Figure 10 shows the introduction of the `mutex` statement. It defines class `MutexBreak`, which is responsible for generating code for the `break` statement inside the `mutex` statement, and `MutexStmt` statement syntax tree class that is responsible for generating code for the `mutex` statement. `MutexStmt` is derived from the `NewStmt1` class, which has `expr1` and `body` variables. Thus, a `MutexStmt` object has `expr1` and `body` variables that point to syntax tree objects for the `mutex` statement argument and its body, respectively. As described in section 2.3.2, the `enter` and `leave` methods are defined so that they change/recover the `break` syntax tree class. A `MutexBreak` object is created if the `break` statement appears inside the `mutex` statement.

The `MutexStmt` `code` method is shown in Figure 11. If a variable pointed to by `expr1` is of the base class `Mutex`, line 6 and the subsequent lines are evaluated. If it is a pointer to a class that is of base class `Mutex`, line 14 and the subsequent lines are evaluated. If the above two conditions are not satisfied, line 20 produces an error message.

The `MutexBreak` `code` method shown in Figure 12 is responsible for generating code that calls the `unlock` method and jumps from the `mutex` statement. To generate code that calls the `unlock` method, the static variable `curEnv` of `BlockStmt` class is checked to traverse outside the block statements of the `mutex` statement.

4 Implementation

Figure 13 shows the organization of the current MPC++ compiler. The frontend processor generates C++ source code with stub routines from the MPC++ source program, or an intermediate code called EXC. The GNU g++ compiler is used as a backend processor to compile C++ source code with stub routines generated by the frontend. The EXC backend processor including the optimizer generates native code from the EXC intermediate code. If no C++ compiler is available on the target machine, the EXC backend processor ports over.

All C++ syntax and extension patterns are described in the `yacc` tool. The lexical analyzer of the frontend processor generates lexical tokens so that the `yacc` tool, an LR parser, can deal with the C++ syntax.

```
$meta MutexStmt::code(OStream os)
{
  mutexp = gensym();
  exitLabel = gensym();
  if (hasBaseType(expr1->getType(), base_typeid(Mutex))) {
    ‘{ Mutex  *$1;
        $1 = ($2).lock();
     }.genCode(os, mutexp, expr1);
    body->genCode(os);
    ‘{ $1: $2->unlock(); }.genCode(os, exitLabel, mutexp);
  } else if (isPointer(expr1->getType())
              && hasBaseType(expr1->getType()->getNext(),
                             base_typeid(Mutex))) {
    ‘{ Mutex  *$1;
        $1 = ($2)->lock();
     }.genCode(os, mutexp, expr1);
    body->genCode();
    ‘{ $1: $2->unlock(); }.genCode(os, exitLabel, mutexp);
  } else {
    errorMessage("illegal mutex operand");
  }
  return GEN_END;
}
```

Figure 11: MutexStmt code Method

```
$meta MutexBreak::code(Ostream os)
{
  BlockStmt     *env = BlockStmt::curEnv;
  while (env) {
      if (hasBaseType(env, typeid(MutexStmt))) {
          ‘{ $1->unlock(); }.genCode(os, ((MutexStmt*)env)->getMutex());
          env = env->getOuter();
      }
  }
  ‘{ goto $1; }.genCode(os, ((MutexStmt*)BlockStmt::curEnv)->getLabel());
  return GEN_END;
}
```

Figure 12: MutexBreak code Method

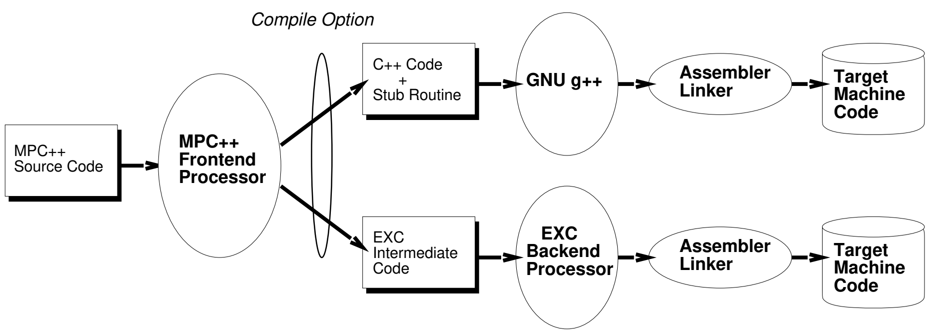

Figure 13: MPC++ Compiler Sytem

The execution of the frontend processor is the same as for the abstract compiler execution. That is, the frontend processor first parses each declaration, such as type, variable, and function declarations including metalevel definitions, and generates a syntax tree object for that declaration with type checking. The syntax tree objects have the `code` and `eval` methods that are responsible for generating the code and interpreting a C++ program, respectively.

5 Evaluation

5.1 C++ Interpreter

The MPC++ frontend processor includes a C++ interpreter to perform metalevel processing. To demonstrate the interpretation speed of the current MPC++ frontend processor implementation, we measured the metalevel execution cost for a `for` loop, a no arguments call/return, "`a = b + c;`," and an object creation. The `for` loop statement involves counter variable initialization, update, and comparison to the limit value.

Table 2 shows the results on Sun Sparc Station 5, 85 MHz. The table contains the execution costs at the metalevel and compares them to the native execution costs. The same programs are used. The interpretation of basic expressions and statements is very slow as currently implemented, because the program is interpreted using syntax tree objects and checks variable types dynamically. However, the object creation cost is only 55 times slower than when using native code.

Because most parts of a metalevel program are described using syntax tree constant expressions, this slow interpretation does not significantly affect actual metalevel processing. To demonstrate this fact, the following subsection measures some metalevel processing costs.

5.2 Metalevel Processing Overheads

We modify the code generation of assignment expressions to evaluate metalevel processing costs. The program shown in Figure 14 is the definition of class `MyAsgnExpr` derived from `AsgnExpr` and a metalevel compound statement to change the syntax tree class of the assignment expression. Two versions of the `code` method are implemented. The first `code` method is a null body, i.e., it just returns to the caller. This program measures the overhead of shifting to metalevel processing. The second method uses a syntax tree expression constant to generate the `printf` call routine before generating the assignment expression.

Table 3: Metalevel Processing Overhead in Assignment Code Modification

	Time (μsec)	Relative
Normal	233	1
Meta(Null)	627	2.7
Meta(Print)	1657	7.1

Table 3 shows the execution time of translating the statement "`a = b + c;` " in three cases, in default compilation and in two metalevel programs. In the default compilation and in the first metalevel program, the translated code is the same as the input statement.

The result shows that it takes about seven times longer than the default compile time to add a simple expression to the generated code of an assignment statement. In terms of absolute execution time, the cost of compiling the same 1,000 statements is about 1.7 seconds compared to a 0.2 seconds default compilation cost. Suppose that the execution time of one metalevel statement takes 200μsec. The cost of compiling bl assignment statements, each of which involves the interpretation of ml metalevel statements will be formulated as follows:

$$(1.7 \ + \ 0.2 \ * \ ml) \ * \ bl \ \ \text{millisecond}$$

Using this formula, for example, the cost of the 10 metalevel statements, that are interpreted to generate the code, is 3.7 seconds for compiling 1,000 lines versus 0.2 seconds.

Note that the compile time of one `mutex` state-

Table 2: Basic Metalevel Processing Cost

	Meta (μsec)	C++ (μsec)	Relative (Meta/C++
`for` statement	167	0.179	933
Call/Return	128	0.095	1,347
`a = b + c`	193	0.061	3,163
Object creation	195	3.530	55

```
$meta class MyAsgnExpr : AsgnExpr {
public:
    int         code(OStream);
};
$meta MyAsgnExpr::code(OStream os)
{
    return GEN_CONT;
}
$meta {
    pushKeyBind("=", EXPR_ASGN, typeid(MyAsgnExpr));
}
```

```
$meta MyAsgnExpr::code(OStream os)
{
    `( (printf("assign expr\n"), ( $1 = $2 ), $1) ).genCode(os, left, right);
    return GEN_END;
}
```

Figure 14: Assignment Code Modification

ment with a null body using the metalevel program, as shown in Section 3, takes about 5 milliseconds.

6 Discussion

6.1 Other Compile-time Metalevel Architectures

OpenC++ Version 2[1] supports a compile-time metalevel architecture. This metalevel architecture differs from the MPC++ architecture. Instead of the introduction of new metalevel classes in MPC++, OpenC++ Version 2 extends the C++ class system so that the user can specify the code generation strategy for each class. This might be the right approach for supporting the compile-time metalevel architecture in a class system. However, C++ has not only object oriented features but also the C language feature. It is necessary to deal with the nested statements shown in this paper and some declaration extensions, such as a global pointer[2, 11]. It is important we understand that such processing is not fit for a class system.

With respect to language syntax extension, OpenC++ has limited capabilities because extensions are class-wise. Since the design of the MPC++ metalevel architecture was carried out after investigating how most existing C++ extensions work[5], the C++ user will naturally extend C++ using MPC++.

Intrigue[7] provides a compile-time metalevel architecture in Scheme to enable the user to define the implementation of the runtime data structure. The user can select the implementation view from several views, i.e., the scheme view, typed view, structure view, and memory view. Because the current MPC++ metalevel architecture assumes that the generated code is in C++ such views are not required. We think, however, that a view for data and control dependencies might be needed to achieve source level optimization.

6.2 Traditional Compilers

The differences between a *compiler & compiler* system and MPC++ are discussed. The compiler & compiler enables language developers to develop a compiler or preprocessor system more easily than from scratch. However, it is still tough to develop a system and modify it to extend the language features because no such modifications and extensions are supported.

Compared with the compiler & compiler system, one of the unique features of the MPC++ metalevel architecture is that abstractions introduced by designers are supplied with the MPC++ metalevel source files. The user can use the right abstractions for demands just by including the files with source files. For example, let us suppose that the `foo.h` file specifies a data parallel computation abstraction while the `bar.h` file specifies a task parallel computation abstraction. If the user wants to use those two abstractions, she/he just writes a program as shown below. Of course, these specifications may not cause side effects for each other.

```
#include "foo.h"
#include "bar.h"
/* data parallel&task parallel program */
```

Some users use a traditional macro processing utility such as *cpp* for C and C++ to introduce their new notations. This, however, notably restricts functionality in the sense that i) it only substitutes a macro expression for a symbol without any grammatical rules, and ii) the compiler can not usually generate the correct error message when the expanded code contains errors.

Compared to the C++ template feature, the syntax tree constant is very similar but has more expressive power. That is to say, a syntax tree constant takes types, expressions, and statements as parameters but a C++ template only takes types as parameters.

6.3 Implementation

As shown in section 5, metalevel processing is slow due to interpretation using syntax tree objects. To speed up the metalevel processing, two options can be considered, i.e., native compilation and on-the-fly byte compilation of the metalevel program. In the compilation of the metalevel program, the metalevel program is compiled to native code which is linked with the MPC++ frontend program. The execution will be fast but the users cannot write metalevel source code in their source programs. That is, the benefits of metalevel programming are sacrificed.

Using the on-the-fly byte compilation technique, the metalevel program is dynamically compiled into byte code, virtual machine code, interpreted by the frontend processor. It will be slower than in the native compilation but faster than in the current implementation.

7 Conclusion and Future Work

We have described the MPC++ metalevel architecture which allows C++ syntax and semantics extension and modification at compile time. The architecture defines the syntax tree classes that represent the parse tree of a program, and defines the MPC++ metaobject protocol. The unique features of the protocol include i) a code generation stack which makes metalevel classes reusable, and ii) a context-dependent code generation mechanism to deal with nested statements in an imperative language. A syntax tree constant, introduced in the MPC++, enables the user easily to define a generation program. Measurements of metalevel processing performance have shown that the current implementation has reasonable execution speed although the C++ interpretation speed is slow. We believe that the metalevel architecture described in this paper is applicable not only to C++ but also to other imperative programming languages.

Applications in MPC++ are manifold. Besides the example given in this paper, we plan to implement data and control parallel constructs[13], realtime and distributed system extension[6]. These programs will constitute the MPC++ metalevel object library (MOL). A global pointer[2, 11], which is a pointer to remote memory, the notion of collection introduced in pC++[4], and a construct for persistent objects have been implemented to test the MPC++ programming capability.

The user may modify the syntax tree objects during the code generation in the current MPC++. Phases, source code transformation and code generation, will be separated in the future MPC++. Facilities for source-to-source code transformation in the metalevel will be also provided.

Acknowledgement

The authors wish to thank Professor Akinori Yonezawa, of the University of Tokyo, and other members of the RWC massively parallel programming language working group for their highly valuable comments on the design of MPC++. Toshiyuki Takahashi, of the University of Tokyo, implemented a part of the MPC++ frontend processor and a runtime system for the Fujitsu AP-1000. The authors would like to thank him.

References

[1] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, pages 285–299, 1995.

[2] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in split-c. In *Supercomputing '93*, 1993.

[3] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without Metaphysics. In *Proceedings of 1984 ACM Symposium on Lisp and Functional Programming*, pages 348–355, 1984.

[4] J. K. Lee Dennis Gannon. Object Oriented Parallelism: pC++ Ideas and Experiments. In *Proceedings of 1991 Joint Symposium of Parallel Processing*, pages 13–22, 1991.

[5] Yutaka Ishikawa. Meta-Level Architecture for Extendable C++. Technical Report TR–94024, RWC, January 1995.

[6] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. In *Proceedings of OOPSLA-90*, pages 289–298, October 1990.

[7] John Lamping, Gregor Kiczales, Luis Rodriguez, and Erik Ruf. An Architecture for an Open Compiler. In *Proceedings of the International Workshop on New Models for Software Architecture '92 (IMSA'92)*, pages 95–106, 1992.

[8] Peter Madany, Panos Kougiouris, Nayeem Islam, and Roy H. Campbell. Practical Examples of Reification and Reflection in C++. In *IMSA'92 International Workshop on Reflection and Meta-Level Architecutre*, pages 76–81, November 1992.

[9] Satochi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hibrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *Fifth ECOOP*, July 1991.

[10] Hideaki Okamura and Yutaka Ishikawa. Object Location Control Using Meta-level Programming. In *Proceedings of 8th ECOOP94*, Lecture Notes in Computer Science 821, pages 299–319. Springer-Verlag, 1994.

[11] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and S. Sekiguti. Distributed Data Structure in Thread-based Programming for a Highly Parallel Dataflow Machine EM-4. In *Proc. of ISCA 92 Dataflow Workshop*, 1992.

[12] T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *OOPSLA'88 Proceedings*, pages 306–315, Sept. 1988.

[13] Gregory V. Wilson and Paul Lu, editors. *Parallel Programming Using C++*. MIT Press, 1996.

[14] Yasuhiko Yokote. The New Mechanism for Object-Oriented System Programming. In *IMSA'92 International Workshop on Reflection and Meta-Level Architecutre*, pages 88–93, November 1992.