TR-94024

Meta-level Architecture for Extendable C++ Draft Document

Yutaka Ishikawa*
ishikawa@rwcp.or.jp

Received 28 Feburary 1995

Tsukuba Research Center, Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305, Japan

Abstract This document describes a meta-level architecture for the MPC++ programming language. The MPC++ meta-level architecture is designed so that programmers can introduce new notation and redefine default code generation with less effort. This capability is very useful for implementing extended C++ languages and some performance and debugging monitors.

The meta-level architecture defines a meta system consisting of the lexical analyzer, parser, and code generator objects that form an abstract compiler, and syntax tree objects that represent the parse tree of a program. Metasystem modification facilities support easy meta-level programming.

*Massively Parallel Software Laboratory

Contents

1 Introduction

User demands to extend programming language semantics or introduce new notation are growing because of the need to adapt a language to paticular application domains such as parallel/distributed environments[2, 4, 5, 12], real-time system[11], and so on. New language notation will make it easier for users to improve program readability and maintainability. Moreover, the compiler may have a chance to generate an efficient code for a high-level abstracted notation.

Some researchers have implemented compilers for new extended programming languages. This works well but requires a great deal of effort. The difficulty, however, is that a base programming language specification becomes hard to follow when that specification is frequently updated as in C++. Other researchers have implemented preprocessors or translators to generate a target language code. This is a simpler approach to implementing an extended language, but, it is difficult to provide a symbolic debugger for such programming languages because the preprocessor cannot generate the information the debugger requires.

Some users use a traditional macro processing utility such as *cpp* for C and C++ to introduce new notation. This approaches, however, results in very restricted functionality in the sense that i) it only substitutes a macro expression for a symbol without any grammatical rules, and ii) the compiler cannot usually generate correct error messages for errors occurring in the expanded code.

To create an extendable programming language, we propose a meta-level architecture for C++. The meta-level architecture has been designed for practical rather than theoretical use. That is, meta-circularity[18] or several levels of meta systems[10, 14] are eliminated. In existing meta-level architectures for object-oriented languages such as ABCL/R[19], ABCL/R2[15], Rbcl[7], AL-1/D[14], and so on, language execution behavior is defined by the set of objects the base-level language can modify. Those systems do not focus on introducing new notation. The architecture proposed here focuses on this capability.

The meta system consists of the lexical analyzer, parser, and code generator objects, which constitute an abstract compiler, and syntax tree objects that represent the parse tree of a program. The user may i) register a new keyword for a syntax pattern selected from predefined patterns so that the lexical analyzer object recognizes the token for the syntax pattern, and ii) define and modify operations over the syntax tree so that the code generation may be modified. Several meta system modification facilities support easy meta-level programming. We believe that this approach is the same as the Metaobject Protocol (MOP) of CLOS[13].

The meta-level architecture is part of MPC++[9] which is an extension to C++. We use this feature to implement other extensions of MPC++ features such as data parallel and real-time programming facilities. In this paper, the proposed meta-level architecture is called the MPC++ meta-level architecture.

The document, which is intended to replace the macro feature proposed in [9], is organized as follows. In section 2, after a summary of the requirements of an extendable C++ language, we present our design philosophy and an overview of the MPC++ meta-level architecture. We also list extension examples that will be developed later in the document using the MPC++ meta-level architecture. In section 3, we extend the C++ language for meta-level programming. Section 4 describes the meta-level system. Section 5 gives readers the MPC++ meta-level protocol which enables users to do easy meta-level programming. Finally, in Section 6, we discuss the current status of the MPC++ meta-level architecture.

2 Design

2.1 Requirements

The compiler compiler enables language developers to develop a compiler or preprocessor system more easily than developing it from scratch. Developing the system and modifying it to extend the language feature, however, is still a challenge. Most extensions do not require major compiler reconstruction, but do require the introduction of a new syntax and semantics which would be implemented by the base language and a stab code. What is needed is a good extension facility that makes it easy to describe the new syntax and semantics. The facility need not be as general as in a compiler compiler, but must not be limited as in a macro processor. Rather, it must allow most C++ extensions to be implemented. The following is a summary of existing C and C++ extensions.

1. New type modifier
   In a parallel system environment, global pointers and related operations have been introduced to access the remote memory area in Split-C[3], EM-C[16], and CC++[2]. To add an additional feature to an object, some modifiers to the class declaration are introduced in RTC++[11], Mentat[6], CHARM++[12], and so on. Of course, the semantics of operations on the modified type, including method invocation, has been newly defined in those extended C++ languages. For example, an `active` modifier of the class declaration in RTC++ has been introduced to describe a server object that is an active entity receiving requests and executing them.

2. New type structure
   In pC++[4], a new structure called *collection* has been introduced to describe data parallel computation. In FO[5], MPC++[9], and so on, a new member declaration part in the class declaration has been introduced to describe parallel/distributed computing. New

invocation mechanisms have been designed for functions declared in these new member declaration parts.

3. New expressions and statements
   A remote invocation expression has been introduced to generate the RPC stab code in languages for distributed computing. Some structured statements for controlling parallel execution are found in COOL[1], Extended-C++[17], CC++, RTC++, and so on.

4. New function capabilities
   New notation has been introduced to give a function new capabilities. An example is a *parallel function* in COOL[1].

5. New modifier for variables
   New notation that declares a parallel variable as a set of storage locations allocated on processors has been introduced in parallel languages. For example, in MPC++, "`int a@[10];`" means that a parallel variable is defined in a one-dimensional array topology processor with a dimension of 10.

2.2 Philosophy

Required extensions are not possible with a macro processor such as `cpp` because the processor must know the syntax and type information. To compile extensions, the processor must parse the program code and handle the parse tree. We employ a meta-level architecture to accomplish this. The semantics of C++ and how the C++ program is compiled are, of course, supplied by the compiler. From the viewpoint of the programmer, the compiler is a meta system of the programming language if the system is written in the same programming language and is visible to and modifiable by the programmer.

We can design many meta-level architectures for C++. Though we can say that a regular C++ compiler written in C++ is a meta system of the C++ program, the compiler does not support a meta-level architecture that modifies and extends C++ features. Rather, such a compiler may be thought of as a traditional boot-strapping compiler.

A meta-level architecture must be elegant in terms of concept and theory. A meta-level architecture, however, is often like a dinosaur in terms of efficiency as a tool for an extendable programming language. What we need is a good meta-level architecture that allows C++ users to modify and extend C++ features easily.

We do not intend to allow users to extend any syntax because good extensions must fit some pattern. Moreover, the system would be too complicated if we allowed any syntax extension the user wanted. Instead, the MPC++ meta-level architecture is designed so that an arbitrary

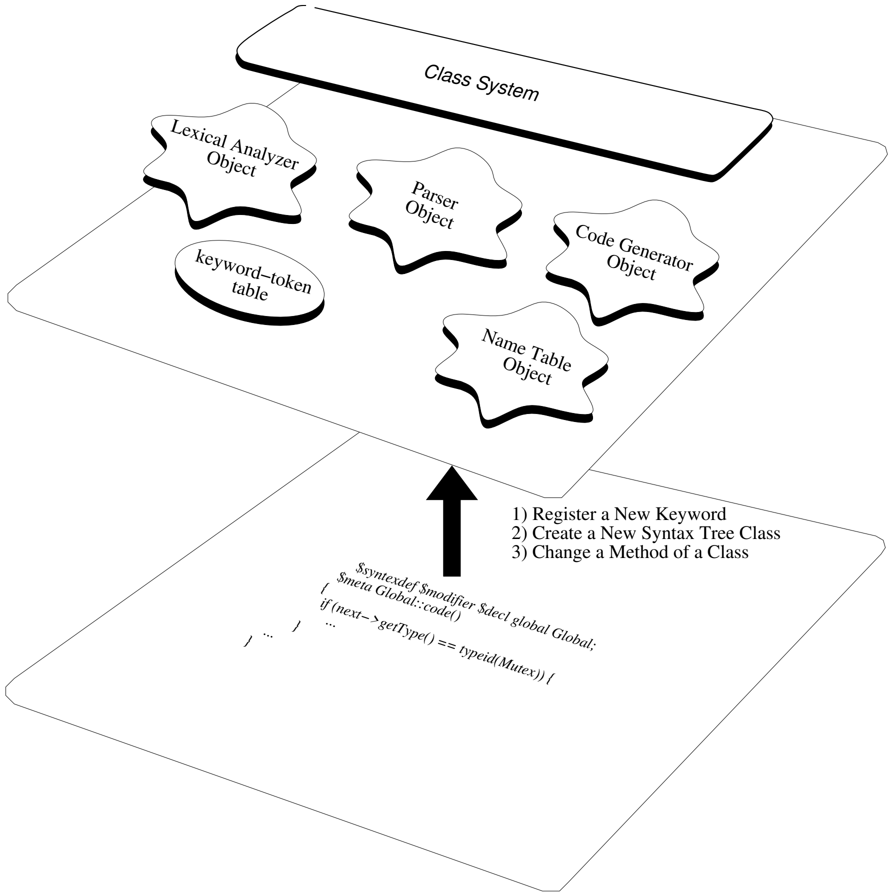

Figure 1: MPC++ Meta-Level Architecture

grammar rule is not dynamically defined but extension patterns are predefined. The extension patterns are designed based on the C++ and existing extended C++ language features summarized above.

The MPC++ meta-level architecture is defined by a set of objects representing a good abstract compiler that the user can modify. Those objects can be described in C++ without any extensions of the class definition system. We did, however, introduce a few extensions to obtain good meta-level programmability. Since many users complain about new keywords for new extensions because they conflict with identifiers, e.g., variable and function names, used in existing programs, we introduced only new keywords that do not conflict with the C++ identifier system.

2.3 Overview

As shown in Figure 1, the MPC++ meta-level architecture defines an abstract compiler and an interface for modifying the compiler using C++.

1. Meta-level abstract compiler
   The abstract compiler consists of a lexical analyzer, parser, code generator, and name table objects. The lexical analyzer object has a keyword table where new keywords are registered by the base-level program. The object takes a C++ program and produces a sequence of tokens. Containing the C++ grammar and grammar for extension patterns, the parser object receives tokens from the lexical analyzer object and creates syntax tree objects based on the grammars. The syntax tree object has methods for checking data type and generating code. The code generator object invokes the code generation method of the syntax tree object. A name table object maintains type, variable, and function names.

2. Meta-level C++ extension
   The following new features are introduced in the meta-level C++ language system to represent the abstract compiler and to enable the C++ user to modify it: i) variable over classes and dynamic object creation from the variable that extends the Run Time Type Information architecture proposed by the ANSI/ISO C++ standard committee, and ii) a parse tree constant that defines a parse tree object statically.

3. Metaobject protocol
   The user can introduce a new syntax and modify the C++ language semantics using the special interface introduced in the C++ language. To allow the user introduce this new syntax, functions that register the new keyword and create the new syntax tree class are provided. The user can program syntax tree methods that transform sources and generate code to define a new syntax or modify the C++ semantics.

2.4 Examples

We use several examples of C++ extensions in the following sections to illustrate how extensions are realized by the MPC++ meta-level architecture. Section 2.4.1 introduces the notion of `atomic` class as an example of type specifier and member modifiers. In section 2.4.2, the implementation of the global pointer exemplifies a declarator modifier. Section 2.4.3 introduces the persistent class to illustrate handling of a structure. Section 2.4.4 introduces remote function invocation and parallel variable access expressions as an example of a new expression. The parallel variable declaration in this section is an example of storage extension capability. Section 2.4.5 introduces a critical region statement as an example of a new statement.

2.4.1 Atomic Class

```
atomic class Buffer {
private:
  char          buf[BUF_SIZE];
  int           in;
  int           out;
public:
  Buffer() { in = out = 0; }
  char  read() when (in ¯ out);?label{ex:when}
  void  write(char)
         when (in ¯ (out + 1)%BUF_SIZE){
          buf[out] = c;
          out = (out + 1) % BUF_SIZE;   }
};
char Buffer::read()
{
    char        ch;
    ch = buf[in];
    in = (in + 1)%BUF_SIZE;
    return ch;
}
```

Figure 2: Atomic Class Example

In an atomic object, only one thread of control is allowed to be executed. Concurrent execution control is achieved by `guard` clauses, each of which can be defined in a method. The method is executed if the expression in the guard clause is true, otherwise, the execution is postponed until the expression becomes true.

Figure 2 shows an example of an atomic class. Class `Buffer` is a bounded buffer where methods `read` and `write` are defined. The `read` method's invocation is suspended if no values have been stored in the buffer. The `write` method's invocation is suspended if the buffer is full. Each guard expression specifies these conditions.

Figure 3 depicts the code generation of the example shown in Figure 2. The generated class has run-time variables `bb` and `sig`. Variable `bb` points to an object of class `Mutex`, which realizes mutual exclusion. Variable `sig` points to an object of class `Signal`, which controls execution. Since the guard expressions are translated into methods `read_guard` and `write_guard`, we call them guard methods.

As shown in Figure 3, methods `read` and `write` are translated as follows:

- Method `enter` of object `Mutex` is called so that execution enters the critical region.
- The guard method is called.
- If the guard method returns the false value, method `suspend` of the `Signal` object is invoked after method `leave` of the `Mutex` object is called. Execution is suspended inside method `suspend`. When execution is resumed, the method is executed again.
- If the guard method returns the true value, the method body is executed. Method `guard_check` is then invoked before the critical region is left.

2.4.2 Global Pointer

Let us consider a global pointer that makes it possible to access remote memory in a distributed memory parallel machine environment. As shown in Figure 4, keyword `global` is introduced to describe the global pointer. Variable `gp` is declared as a global pointer in line 1. Although the operations of a global pointer use the same semantics as an ordinal C++ pointer, the code generation is different.

Figure 5 outlines the code generation of the program in Figure 4. The representation of `gp` is structure `_globalStorage`. All operations of the global pointer are transformed to calling libraries, e.g. calling of `_globalAddrSet` for the assignment expression.

2.4.3 Persistent Object

This section presents a simple persistent object. As shown in Figure 6, an object bound in a `permanent` variable is a persistent object residing in the secondary storage area. The persistent object is loaded when the program is executed.

Figure 7 illustrates the code generation of the program in Figure 6. New functions `_readPO_A` and `_write_PO_A` are generated to read data for the object from secondary storage and to save the object data to storage, respectively. New functions `_setupPO_A_a1` and `_endupPO_A_a1` are produced for initialization and finalization of the `a1` persistent object.

2.4.4 Remote Expression

Let us introduce a parallel variable and remote function invocation. Line 2 in Figure 8 declares parallel variable `bar`, which exists on processors. In line 5, `bar`'s value on processor 2 is referred to.

Function invocation followed by `@` in line 5 is a remote function invocation expression. In the expression, the `foo` function is invoked on processor 10.

Figure 9 shows the translated code from the example in Figure 8.

2.4.5 Mutex Statement

To set up the critical region in parallel processing, the programmer creates the `Mutex` class, which has `enter` and `leave` methods. The critical region is programmed in such a way that method `enter` must be invoked before the critical region is entered, and method `leave` must be invoked on exiting the critical region. Sample code is shown below.

```
class Buffer {
private:
  char          buf[BUF_SIZE];
  int           in;
  int           out;
  Mutex         bb;
  Signal        sig;
public:
  Buffer() { in = out = 0; }
  int   read_guard() { return in != out; }
  int   write_guard()
        { return in != (out + 1)%BUF_SIZE; }
  void  write(char ch) {
         retry:
          bb.enter();
          if (write_guard() != TRUE) {
            bb.leave(); sig.suspend();
            goto retry;
          } else {
            // write body is here
            guard_check();
            bb.leave();
          }
        }
  void  guard_check() {
          if (sig.isWaiting()) {
            sig.allresume();
          }
        }
};
char Buffer::read()
{
    char        ch;
 retry:
    bb.enter();
    if (read_guard() != TRUE) {
        bb.leave(); sig.suspend();
        goto retry;
    } else {
        // read body is here
        guard_check();
        bb.leave();
        return ch;
    }
}
```

Figure 3: Code Generation for Atomic Class Example

```
int * global    gp;
int             li;

main()
{
    gp = &li;
    *gp = 10;
    foo(*gp + 2);
}
```

Figure 4: Global Pointer Example

```
struct _globalStorage   gp;
int                     li;

main()
{
    _globalAddrSet(&gp, (void*)&li);
    _globalPut(&gp,10);
    foo(_globalGet(&gp) + 2);
}
```

Figure 5: Code Generation for Global Pointer Example

```
class A { int i; float f; };
permanent A     a1;
permanent A     a2;
```

Figure 6: Persistent Object

```
class A {
    friend      _readPO_A(char*, A*);
    friend      _writePO_A(char*, A*);
    int         i;
    float       f;
};
A       a1;
A       a2;
_readPO_A(char *name, A *obj)
{
    pf_seek(name);
    pf_read(sizeof(int), (void*) &obj->i);
    pf_read(sizeof(float), (void*) &obj->f);
}
_writePO_A(char *name, A *obj)
{
    pf_seek(name);
    pf_write(sizeof(int), (void*) &obj->i);
    pf_write(sizeof(float), (void*) &obj->f);
}
_startup_PO_A_a1() { _readPO_A("a1", &a1); }
_endup_PO_A_a1() { _writePO_A("a1", &a1); }
_startup_PO_A_a2() { _readPO_A("a2", &a2); }
_endup_PO_A_a2() { _writePO_A("a2", &a2); }
```

Figure 7: Code Generation for Persistent Object

```
void    foo(int, int);
int     bar@[10];
sub()
{
    return foo(1, 2)@[10] + bar@[2];
}
```

Figure 8: Remote Expression

```
void    foo(int, int);
int     bar;
sub()
{
    ...;
    _rpc(foo, 10, 1, 2) + _rmc(&bar, 2);
    ...;
}
```

Figure 9: Code Generation for Remote Expression

```
 1 Mutex   mm;
 2 void sub()
 3 {
 4     mutex(mm) {
 5         if (/* ... */) break;
 6         for (/**/;/**/;/**/) {
 7             /**/
 8             if (/* ... */) break;
 9             if (/* ... */) goto l1;
10         }
11         if (/* ... */) return;
12     l1:
13         if (/* ... */) goto l2;
14     }
15     /* ... */
16   l2:
17     return;
18 }
```

Figure 10: Mutex Example

```
Mutex   mm;
sub()
{
    mm.enter();
    /* critical region here */
    mm.leave();
}
```

One risk of the above program is that the critical region would remain unfinished if the programmer forgot to invoke method `leave`. Here, we introduce a new statement `mutex` to realize the critical region.

Figures 10 and 11 show a program example using the `mutex` statement and the code generation for the example. In the example, the `break` statement in line 5 exits from the `mutex` statement, and the `break` statement in line 8 exits from the `for` loop statement. The `goto` statement in line 9 jumps to the `l1` label defined within the `mutex` statement, and the `goto` statement in line 13 jumps to outside the `mutex` statement. Thus, the code of the `goto` statement in line 13 includes the exit code of the `mutex` statement, i.e., the invocation of `mm.leave` method in line 13. If a `return` statement appears within the `mutex` statement as in line 11, the code generation also includes the exit code of the `mutex` statement as in line 11.

```
Mutex   mm;
void sub()
{
    mm.enter(); {
        if (/* ... */) goto LID001;
        for (/**/;/**/;/**/) {
            /**/
            if (/* ... */) break;
            if (/* ... */) goto l1;
        }
        if (/* ... */) { mm.leave(); return; };
    l1:
        if (/* ... */) { mm.leave(); goto l2};
    }
  LID001:
    mm.leave();
    /* ... */
  l2:
    return;
}

```

Figure 11: Code Generation for Mutex Example

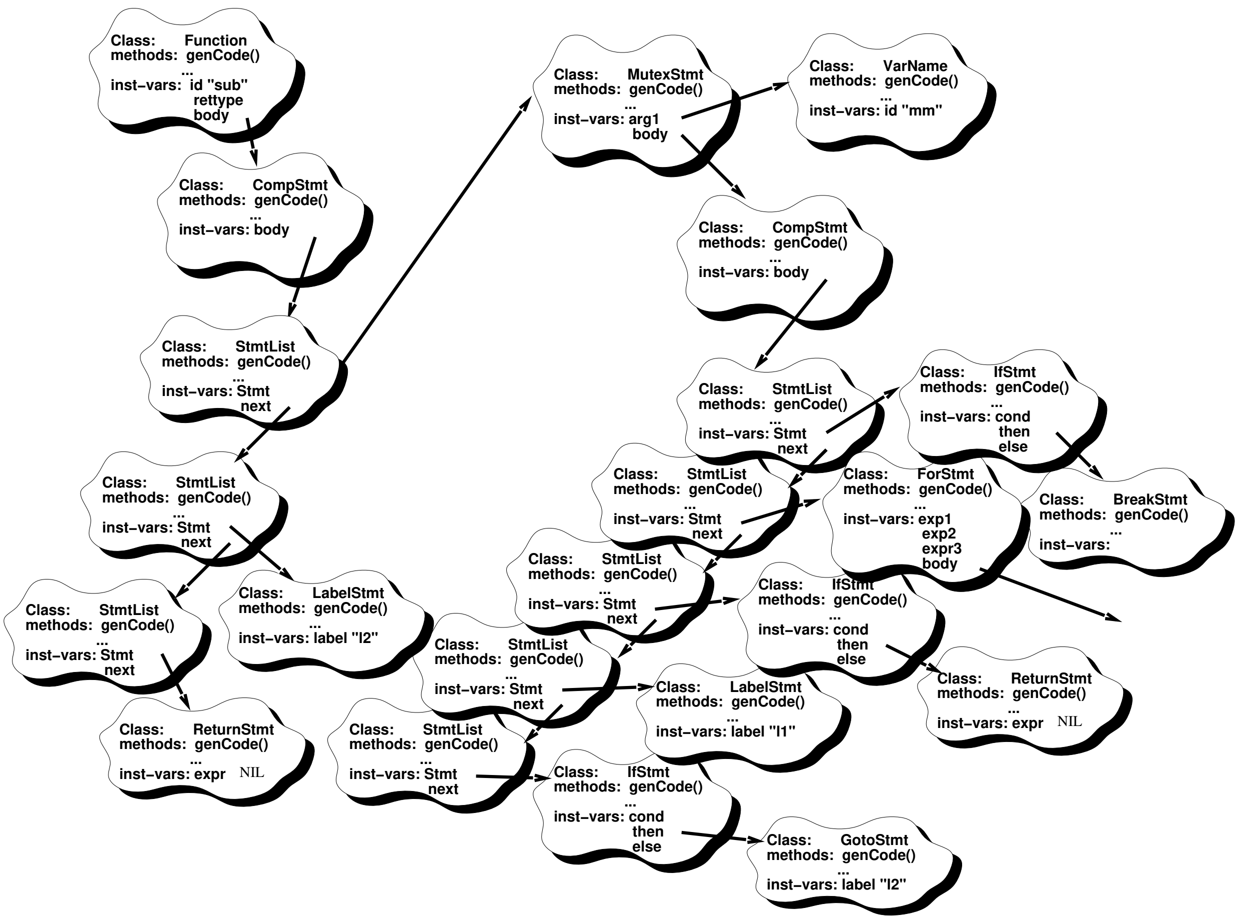

Figure 12: A Syntax Tree

3 Meta-level C++ Extension

Some C++ extensions are introduced at the meta-level to represent and handle the syntax tree object. Since those extensions are only meta-level features, C++ base-level programming is not affected.

3.1 Variable over Classes

If we implemented a naive parser, the program in Figure 10 would be transformed to the syntax tree shown in Figure 12. Each node of the tree is represented by a syntax tree object. Syntax tree object `ReturnStmt` in this figure means that execution exits from the `mutex` statement and returns from the `sub` function. These semantics are different from the normal return statements. The parser should generate a different syntax node, e.g. `MutexReturnStmt`, with respect to the semantics. These different semantics originate from structural behavior (or context), i.e., whether or not they are inside the `mutex` statement.

Statically, the parser knows neither all contexts derived from new statements nor syntax tree classes because the user may introduce new contexts and classes using the MPC++ meta-level feature. Since C++ requires concrete class definition before creating an instance object of a class, we cannot program the parser so that the syntax tree object is created dynamically from a class that has been created with respect to the context. If we have a variable over classes, the syntax tree object is created using the variable that is bound to a particular class according to the context.

We follow and extend the Run Time Type Information architecture in the draft C++ standard. That is, the special class `type_info` and the `typeid` operator are builtin to keep information on a type and to refer to a type_info that represents the type, respectively. In addition to operator `==`, operator `<=` is defined in the class to check the subclass relation. Moreover, the `base_typeid` operator is introduced to represent a base-level type.

In the following example, `anyClass` is a variable over types.

```
class Return;
const type_info *anyClass;
```

We need a facility to create an object from a type variable bound to the type identifier of a class. Unfortunately, since the current draft C++ standard does not specify the facility, we introduce a new expression called `dynamic_new` for that purpose. The following is an example:

```
class Return : Stmt {/*...*/};
class MutexReturn : Return {/*...*/};
foo()
{
    const type_info     *returnClass;
    Return              *returnNode;

    returnClass = typeid(Return);
    returnNode = dynamic_new returnClass; // create an object of class Return
    returnClass = typeid(MutexReturn);
    returnNode = dynamic_new returnClass; // create an object of class MutexReturn
}
```

3.2 Syntax Tree Constant

A syntax tree object may be defined as a constant value, called a *syntax tree constant*. An expression surrounded by ‘(and) represents a syntax tree object for the expression. A statement surrounded by ‘{ and } represents the statement syntax tree object.

syntax-tree-constant:
 ‘(*expression*)
 ‘{ *statement* }
 ‘[*parameter-declaration-list*]

The following is an example:

```
‘( 1 + 2 + 3 )
‘{ foo() { return i; } }
```

The user may use variables in a syntax tree constant. 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. Since a syntax tree constant is an object derived from class SynObj, methods of that class may be invoked.

```
‘($1 + $2).genCode(10, 20);
‘{ struct $1 { int i; }; }.genCode("foo");
‘{ int $1(int i) { return i; } }.genCode("bar");
‘{ $$1 $2; }.genCode(new BasicType(TYPE_INT), "a");
‘{ $$1 $2[$3]; }.genCode(base_typeid(float), "f", 10);
```

If the above examples are executed by invoking method genCode, the following code is generated:

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

4 Meta-level Objects

As shown in Figure 13, the abstract compiler at the meta-level consists of i) the lexical analyzer, parser, and code generator objects making up the abstract compiler, ii) syntax tree

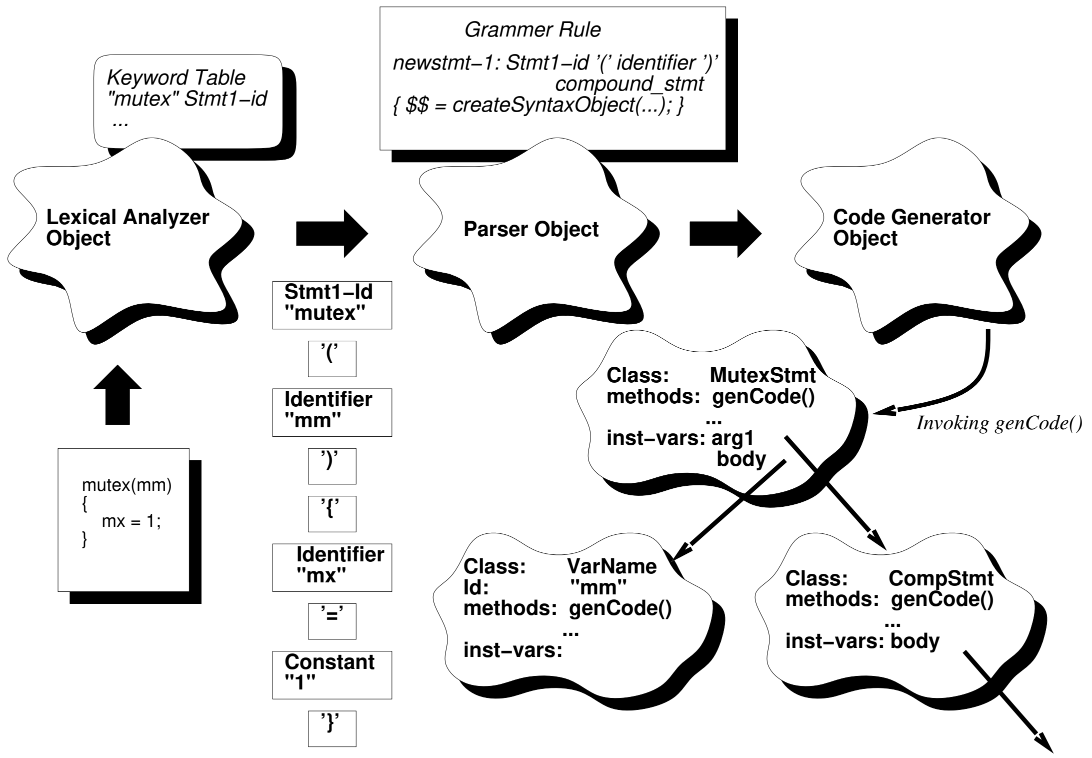

Figure 13: Abstract Compiler Overview

objects representing the parse tree of a program, and iii) name table objects that manage local/global type, variable, and function names and are visible from the syntax tree objects. Those objects are described in this section.

4.1 Lexical Analyzer Object

The lexical analyzer takes a C++ source program and produces tokens. Users can register a new keyword with any of the predefined patterns, which are carefully designed to accommodate most existing extended C++ languages. When the lexical analyzer reads a newly registered keyword, it produces a token defined by the syntax pattern and the class name associated with the keyword.

Figure 14 shows the user interface of the lexical analyzer object. The user invokes method `bindKeyword` to register a new keyword. The method's arguments are the keyword name, token type, and the type identifier of a syntax tree object. The definition of the token type is given in an appendix.

To achieve context dependent processing in the abstract compiler, the association of keywords and type identifiers in the keyword table is dynamically modified. Methods `pushKeywordTable` and `popKeywordTable` are responsible for handling those versions of the keyword table.

```
class LexicalAnalyzer {
    friend class Parser;
private:
    TokenType   getToken(typeid &clname); // for parser object
public:
    int         bindKeyword(char *name, TokenType tt, typeid clname);
    void        pushKeywordTable();
    void        popKeywordTable();
};
extern LexicalAnalyzer  *lex;
```

Figure 14: Lexical Analyzer Object

4.2 Parser Object

In addition to the C++ grammar rules, the parser object has grammar rules for predefined syntax patterns shown in appendix A. For example, the grammar rule with token `NEW_STMT1` is the new statement which takes one argument and compound statement in Figure 13. The parser object is the main routine in the abstract compiler for the following processing.

1. The parser object invokes method `getToken` of the lexical analyzer object to receive a token. If the token is a keyword, the type identifier of the parse tree class for that keyword is also received.

   (a) For a statement token, the procedure is as follows: The parser object creates an object of the parse tree class using the received type identifier. The created object is called a *top environment* object. The object method `cntxtEnter` is invoked so that the keyword table kept in the lexical analyzer object can be modified in method `bind` called by `cntxtEnter`.

   (b) For an expression operator token, the procedure is as follows: Syntax trees of the expression operands are first produced. The syntax tree class representing the expression is selected using operand types and the operator token.

   The parser object keeps a database of the parse tree classes and their operand types. As shown in Figure 15, methods `regUnaryOp` and `regBinaryOp` are provided to register the database entry.

   (c) For a specifier or declarator token, method `declstruct` is invoked after the syntax tree object is obtained to give a modifier object a chance to modify the original structure.

   (d) For a member function/storage token, method `memberModify` is invoked after the syntax tree object is obtained to give a modifier object a chance to modify the original member function/storage.

   (e) In all other cases, the parser object creates a parse tree class object using the received type identifier.

```
class Parser {
public:
    int         regUnaryOp(TokenType op, typeid operand, typeid newUnaryOp);
    int         regBinayOp(TokenType op, typeid left, typeid right,
                           typeid newBinaryOp);
};
extern Parser   *parser;
```

Figure 15: Parser Object

2. The parser object continues to invoke `getToken` of the lexical analyzer object and generates a syntax tree object according to the grammar rule. The parser object then invokes method `cntxtLeave` of each top environment object produced.

3. The parser object invokes method `genCode` of the code generator object with the syntax tree object as the argument.

4.3 Code Generation Object

The code generation object is responsible for generating the code. Method `genCode`, whose argument is a syntax tree object, is invoked by the parser object. The code generation object generates target code by invoking the code generation method of the syntax tree object.

4.4 Name Table Object

A name table object keeps all information on types, functions, and variables names. The name table object that keeps global names is called the global name table object. Several name table objects, called local name table objects, are created to handle the C++scope rule. The meta-level variable `globalNameTable` points to the global name table object. The meta-level variable `curNameTable` points to a local name table object. The user cannot change the C++ scope rule in this version of the meta-level architecture.

4.5 Syntax Tree Object

A syntax tree object represents a type specification, storage/function declaration, expression, or statement. Figure 16 shows the class hierarchy of syntax tree classes. A syntax tree class is derived from class `SynObj`. When a new syntax is introduced using the meta-level facility, a new class representing the syntax is created. This class must be a subclass of the `NewSpecDeclModifier`, `NewMemModifier`, `NewStructDecl`, `NewSpec`, `NewStmt1` to `NewStmt6`, `NewUnaryExpr`, `NewBinaryExpr`, or `NewTrinaryExpr` class in Figure 16. Declaration of syntax tree classes is given in appendix B.

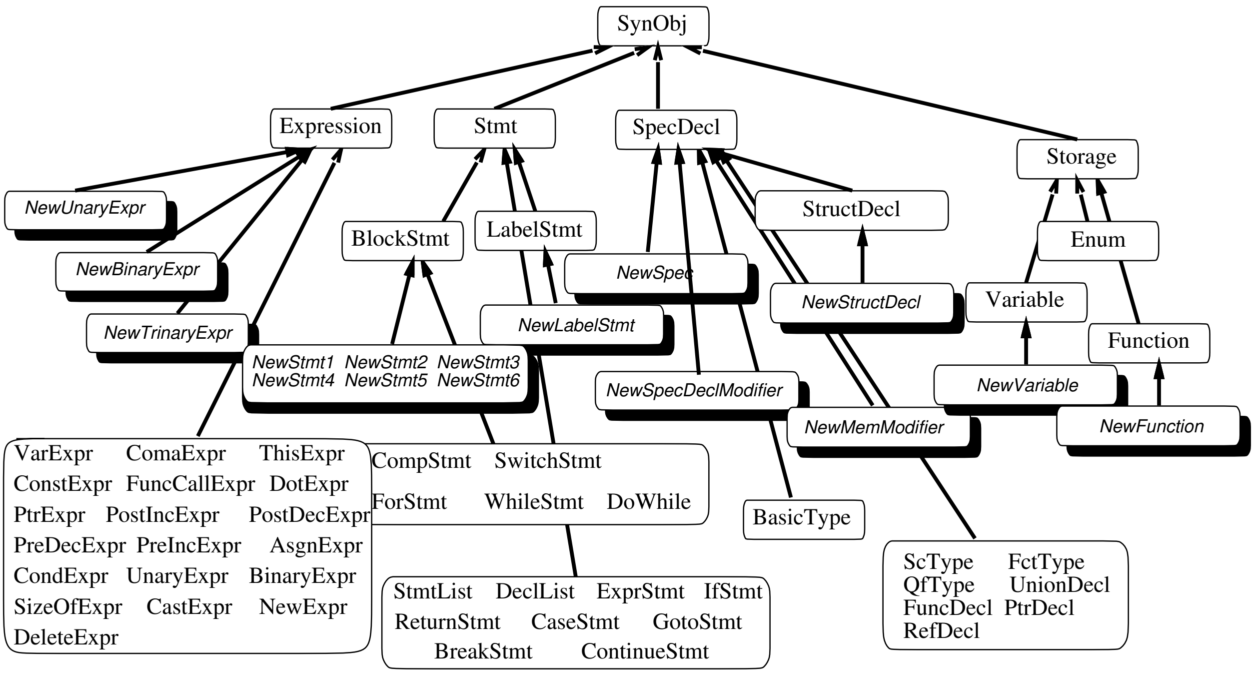

Figure 16: Parse Tree Classes

4.6 Examples

MPC++ meta-level objects are used to create extension examples presented in Section 2.4. Although programs in the rest of this section cannot be used exactly as presented, they nevertheless will help the reader to understand the architecture. Correct programming in MPC++ MOP will appear in the next section.

4.6.1 Atomic Class

Let us implement the atomic class described in Section 2.4.1. Figure 17 shows the declaration of classes `Atomic` and `Guard`. Class `Atomic` represents the syntax tree object for `atomic` specifier modifier. Class `Guard` is declared to represent the `guard` clause syntax tree object. To allow the new keywords `atomic` and `guard` to be handled in the lexical analyzer object, the statements in lines 18 and 18 must be executed.

Class `AtomicReturn` represents the return statement within a method of the atomic class. The code for `return` for the atomic class must be generated where method `leave` of class `Mutex` is invoked. The program in Figure 18 shows that class `AtomicReturn` is defined and keyword `return` and the class are associated in line 7.

Method `declStruct` of class `Atomic` is defined below. The method is invoked when a class is declared with the `atomic` modifier.

```
class Atomic : public NewSpecDeclModifier {
private:
    virtual SynObj      *memberModify(Function *);
    virtual SynObj      *declStruct();
    void                bind();
    virtual SynObj      *storage(Symbol *name) { return next->genStorage(name); }
    virtual SynObj      *type() { return next->getType(); }
public:
    static void         init();
};
class Guard : public NewMemModifier {
public:
    void                setFuncName();
    SynObj              *getFuncName();
};
void Atomic::init()
{
    lex->bindKeyword("atomic", SPEC_MODIFIER, Atomic);
    lex->bindKeyword("guard", MEMBER_MODIFIER, Guard);
}
```

Figure 17: Classes Atomic and Guard

```
class AtomicReturn : public ReturnStmt {
private:
    int                 code() { `{ bb.leave(); }.genCode(); return GEN_CONT; };
};
void Atomic::bind()
{
    lex->bindKeyword("return", RETURN, AtomicReturn);
}
```

Figure 18: Return Statement inside the Atomic Member Method

```
SynObj  *Atomic::declStruct()
{
    AccDecl             *ad;
    SpecDecl            *decl;
    NewMemModifier      *mod;
    StructDecl          *st;
    BaseSpec            *bspec;
    StructDecl          *bdecl;
    int                 hasBases = FALSE;
```

```
10     if (!(typeid(next) <= typeid(StructDecl*))) {
11         errorMessage(lineno, "atomic does not modify structure");
12         return NIL;
13     }
14     st = (StructDecl*) next;
15     if ((bspec = st->getBaseSpec())) {
16         hasBases = TRUE;
17         while (bspec) {
18             bdecl = bspec->getDecl();
19             if (!bdecl->searchModifier(typeid(Atomic*))) {
20                 errorMessage(lineno, "the base class is not atomic class\n");
21                 return NIL;
22             }
23             bspec = bspec->getNext();
24         }
25     } else {
26         st->addPrivateMember(`{Mutex bb;});
27         st->addPrivateMember(`{Signal sig;});
28     }
30     for (ad = st->getAccDecl(); ad; ad = ad->getNext()) {
31         for (decl = ad->getDecl(); decl; decl = decl->getNext()) {
32             if (typeid(decl) == typeid(Function*)) {
33                 if ((mod = decl->modifier())
34                     && typeid(mod) == typeid(Guard*)) {
35                     /* generating a guard method */
36                     fname = new Symbol(decl->getId(), "_guard");
37                     ((Guard*)mod)->setFuncName(fname);
38                     fp = new Function(fname, decl->numArgs(), decl->getArglist());
39                     fp->addStatement(mod->getBody());
40                     st->addPrivateMember(fp);
41                 }
42             }
43         }
44     }
45     if (hasBases == FLASE) {
46         st->addPrivateMember(`{ void guard_check() {  if (sig.isWaiting()) {
47                                                          sig.allresume();
48                                                        }
49                                                     } } );
50     }
51     return next;
52 }
```

Method `declStruct` of class `Atomic` performs the following tasks: i) checks whether or not it modifies the class declaration in lines 10 to 13, ii) modifies the class declaration in lines 26 to 27 so that it contains member variables `bb` and `sig`, as in Figure 3, iii) produces methods for guard clauses by traversing all member declarations of the class in lines 30 to 44, and iv) generates method `guard_check` in lines 46 to 49.

Method `memberModify` of class `Atomic` is defined below. It is invoked when a member method pointed by the argument is declared.

```
class Global : public SpecDeclModifier {
private:
    virtual SynObj      *storage();
    virtual SynObj      *type();
public:
    static void         init();
};
SynObj* Global::storage()
{
    return `{ struct _globalStorage; }.genStorage();
}
SynObj* Global::type()
{
    return `{ struct _globalStorage; }.getType();
}
void Global::init()
{
    lex->bindKeyword("global", DECL_MODIFIER, Global);
}
```

Figure 19: Class Global Declaration

```
 1 SynObj  *Atomic::memberModify(Function *fp)
 2 {
 3     FuncDecl    *fdecl = fp->getDecl();
 4     SpecDecl    *mod;
 5
 6     if (mod = fdecl->searchModifier(typeid(Guard*))) {
 7         IfStmt          *ifStmt = new IfStmt();
 8         Expr            *expr;
 9         Stmt            *stmt;
10         StmtList        *top;
11         CallParam       *param;
12
13         top = new StmtList(`{ retry: bb.enter(); });
14         param = fdecl->getParamList();
15         expr = new FuncCallExpr(((Guard*)mod)->getFuncName(), param);
16         expr = new BinaryExpr(OP_NEQ, expr, `(1));
17         ifStmt->setCond(expr);
18         ifStmt->setThenPart(`{bb.leave(); sig.suspend(); goto retry;});
19         stmt = fp->getBody();
20         stmt->addStmt(`{guard_check(); bb.leave();});
21         ifStmt->setElsePart(stmt);
22         top->addStmt(ifStmt);
23         fp->setBody(top);
24     }
25     return fp;
26 }
```

In method `memberModify` of class `Atomic`, if the method denoted by the argument is modified by the `guard` clause in line 6, the method body is modified as in Figure 3.

4.6.2 Global Pointer

To implement the global pointer, we have to define i) the storage generation for a global pointer variable and ii) the code generation for expressions involving global pointers. Class `Global` derived from `SpecDeclModifier` is defined to represent the global pointer in Figure 19.

The `storage` method of class `Global` defines the storage definition. Here, structure `_globalStorage`

```
 1 class GlobalPtrRefExpr : public PtrRefExpr {
 2 private:
 3     virtual int         code();
 4     virtual SynObj      *type();
 5 };
 6 GlobalPtrRefExpr::code()
 7 {
 8     SynObj      *dtype = next->getType();
 9     `( _globalGet(&$1) ).genCode(next);
10 }
11 SynObj* GlobalPtrRefExpr::type()
12 {
13     SynObj      *dtype = next->getType();
14     return dtype->deRef(dtype);
15 }
16 {
17     parser->regUnaryOp(UNARYEXPR, POINTER, Global, GlobalPtrRefExpr);
18 }
```

Figure 20: Class GlobalPtrRefExpr Declaration

is used as the meta-level (or implementation) representation. Method `type` indicates the meta-level data type. After invoking the statement in line 18, the program below on the left is translated into the program on the right.

```
int * global    gp;                                struct _globalStorage   gp;
```

`GlobalPtrRefExpr` derived from `PtrRefExpr` is defined to represent the global pointer reference in Figure 20. After invoking the statement in line 17, the program below on the left is translated into the program on the right.

```
int i = *gp;                                       int    i = _globalGet(&gp);
```

However, since the data type of variable `gp` is the global pointer in line 9 of the `Global` method `code`, the argument type of `_globalGet` is the address of the global pointer that againsts the formal type, the address of the `_globalStorage` structure. We therefor need the capability to change representation so that the syntax tree object pointed to by the `next` variable in line 9 represents the meta-level (or implementation) data type. Method `metaRep` is provided in class `SpecDecl` to accomplish this.

```
GlobalPtrRefExpr::code()
{
    SynObj      *dtype = next->getType();
    next->metaRep();
    `( _globalGet(&$1) ).genCode(next);
}
```

Figure 21 defines the syntax tree object representing the global pointer assignment expression.

```
class GlobalAsgnExpr : public AsgnExpr {
private:
    virtual int         code();
    virtual SynObj      *type();
};
GlobalAsgnExpr::code()
{
    SynObj      *dtype = left->getType();
    left->metaRep();
    `( _globalAddrSet(&$1, (void*)($2)) ).genCode(left, right);
}
SynObj* GlobalAsgnExpr::type()
{
    SynObj      *dtype = left->getType();
    return dtype->deRef(dtype);
}
void initGlobalAsgnExpr()
{
    parser->regBinaryOp(BINARYEXPR, AsgnExpr, Global, Global, GlobalAsgnExpr);
}
```

Figure 21: Class GlobalAsgnExpr Declaration

4.6.3 Persistent Object

Class `Permanent` derived from class `SpecDeclModifier` is defined to implement the persistent object presented in section 2.4.3.

```
 1 class Permanent : public SpecDeclModifier {
 2 private:
 3     void                genPOfunc(StructDecl *decl);
 4     void                genInitPO(Symbol *name, StructDecl *decl);
 5     virtual SynObj      *storage(Symbol *name);
 6     virtual SynObj      *type() { return next->getType(); }
 7     static Table        *initTable;
 8 public:
 9     static void         init();
10 };
11 void Permanent::init()
12 {
13     lex->bindKeyword("permanent", SPEC_MODIFIER, Permanent);
14 }
```

The `storage` method of class `Permanent` is invoked when a variable is declared with the `permanent` modifier. We assume that the `next` variable of a `Permanent` object points to a structure declaration object. Variable `initTable` keeps the structure's identifiers. Structures stored in `initTable` have been processed as persistent objects. If a structure's identifier has not been stored in the `initTable`, function `genPOfunc` is invoked in line 4 to produce run-time library functions to read/write a persistent object from/to the secondary storage. Function `genInitPO` is invoked in line 8 to generate run-time library `setup` and `endup` functions for the variable denoted by variable `name`.

```
SynObj* Permanent::storage(Symbol *name)
{
    Identifier  *id = next->getId();
    if (!initTable->find(id)) {
        genPOfunc((StructDecl*) next);
        initTable->enter(id);
    }
    genInitPO(name, (StructDecl*) next);
    return next->genStorage(name);
}
```

`genPOfunc` generates a run-time library routines to read/write a persistent object from/to secondary storage. All that is required is one argument, `st`, pointing to the syntax tree object of a structure declaration. First, the syntax tree object of the function whose name is `_readPO_`, followed by the structure's name, is created. The function body is generated in lines 15 to 33. The loop statement in line 22 visits all access declarators, such as `public`, `private`, and `protected`, and variable `ad` points to an `accDecl` object that keeps member declarators. The loop statement in line 23 traverses all member declarators in the `accDecl` object pointed to by `ad`. If a member declarator represents a variable, then a read function invocation statement is generated in line 26 to initialize the variable by reading its stored value from secondary storage.

Lines 36 to 59 define the generation code of a function that writes the persistent object to secondary storage.

```
 1 void Permanent::genPOfunc(StructDecl *st)
 2 {
 3     Symbol      *fname;
 4     Function    *fp;
 5     CompStmt    *cs;
 6     SynObj      *syn;
 7     AccDecl     *ad;
 8     BaseSpec    *bspec;
 9     StructDecl  *decl;
10     fname =  new Symbol("_readPO_", st->getId());
11     syn = `{friend void $1(char *, $$2*);}.genSynObj(fname, st);
12     st->addMember(syn);
13     cs = new CompStmt;
14     fp = new Function(fname, 2, `[char *name, $$1*obj].genSynObj(st), cs);
15     syn = `{ pf_seek(name); }.genSynObj();
16     cs->addStatement(syn);
19     bspec = st->getBaseSpec();
20     decl = st;
21     do {
22         for (ad = decl->getAccDecl(); ad; ad = ad->getNext()) {
23             for (decl = ad->getDecl(); decl; decl = decl->getNext()) {
24                 if (decl->isStorage()) {
25                     syn = `{ pf_read(sizeof($1), (void*) &obj->$2)}
26                             .genSynObj(decl->getType(), decl->getId());
27                     cs->addStatement(syn);
28                 }
29             }
30         }
31     } while (bspec
32              && (bspec = bspec->getNext())
33              && (decl = bspec->getDecl()));
34     fp->genCode();
```

```
    fname =  new Symbol("_writePO_", st->getId());
    syn = `{friend void $1(char *, $$2*);}.genSynObj(fname, st);
    st->addMember(syn);
    cs = new CompStmt;
    fp = new Function(fname, 2, `[char *name, A *obj], cs);
    syn = `{ pf_seek(name); }.genSynObj();
    cs->addStatement(syn);

    bspec = st->getBaseSpec();
    decl = st;
    do {
        for (ad = decl->getAccDecl(); ad; ad = ad->getNext()) {
            for (decl = ad->getDecl(); decl; decl = decl->getNext()) {
                if (decl->isStorage()) {
                    syn = `{ pf_read(sizeof($1), (void*) &obj->$2)}
                           .genSynObj(decl->getType(), decl->getId());
                    cs->addStatement(syn);
                }
            }
        }
    } while (bspec
             && (bspec = bspec->getNext())
             && (decl = bspec->getDecl()));
    fp->genCode();
}
```

```
void Permanent::genInitPO(Symbol *name, SynObj *decl)
{
    Symbol      *fname;
    Symbol      *cname;

    fname = new Symbol("_startup_PO_", decl->getId(), "_", name);
    cname = new Symbol("_readPO_", name);
    funcBegin(fname, 0);
    `{ $1($2, &$3); }.genCode(cname, name->asString(), name);
    funcEnd();

    fname = new Symbol("_endup_PO_", decl->getId(), "_", name);
    cname = new Symbol("_writePO_", name);
    funcBegin(fname, 0);
    `{ $1($2, &$3); }.genCode(cname, name->asString(), name);
    funcEnd();
}
```

4.6.4 Remote Expression

In Figure 22, Classes `Parallel` derived from `StorageModifier` and `RemoteExpr` derived from `BinaryExpr` are introduced to implement a parallel variable and remote function invocation, respectively. After executing function `initRemoteExpr`, the lexical analyer recognizes the `@` keyword as both a storage modifier and postfix expression.

The code generation for the `@` postfix expression is defined in the `RemoteExpr`'s method `code` shown below. This is a simple method in the sense that the _rpc function invocation is generated if the left expression is a function invocation expression, and the _rmc function invocation is generated if the left expression is a parallel variable.

```
class Parallel : public StorageModifier {
private:
    virtual SynObj      *storage(Symbol *name) { return next->storage(name); }
    virtual SynObj      *type() { return this; }
public:
    static void         init();
};
void Parallel::init()
{
    lex->bindKeyword("@", STORAGE_MODIFIER, Parallel);
}
class RemoteExpr : public BinaryExpr {
private:
    virtual int         code();
    virtual SynObj      *type() { return next->getType(); }
public:
    static void         init();
};
void RemoteExpr::init()
{
    lex->bindKeyword("@", NEW_EXPR_POSTFIX_DIMMENSION, RemoteExpr);
}
```

Figure 22: Classes Parallel and RemoteExpr

```
RemoteExpr::code()
{
    SynObj      *dtype = NIL;

    if (typeid(left->getType()) == typeid(Parallel*)) {
        if (typeid(left) <= typeid(FuncCallExpr*)) {
            `( _rpc($1, $2, $3) ).genCode(left->getFunc(), right,
                                          left->getParam());
        } else if (typeid(next) <- typeid(VarName)) {
            `( _rmc(&$1, $2) ).genCode(left, right);
        } else {
            // error
        }
    }
    return GEN_END;
}
```

4.6.5 Mutex Statement

Let us implement the mutex statement described in Section 2.4. Class `MutexStmt` representing the mutex statement is defined below. Because the mutex statement takes one argument, the class is derived from class `NewStmt1`.

```
class MutexStmt : public NewStmt1 {
private:
    Label       *exitLabel;
    SynObj      *mutexp;
    void        bind();
public:
    int         code();
    Status      genEnter();
    Status      genLeave();
    MutexStmt(SynObj *arg, Stmt *bk) : Stmt(arg, bk) { }
    static void init();
};
void MutexStmt::init()
{
    lex->bindKeyword("mutex", NEWSTMT_1, MutexStmt);
}
```

Method `init` of class `MutexStmt` registers the new statement `mutex`, which takes one argument and one compound statement, into the lexical analyzer object. The `MutexStmt` `bind` method is defined below. The method replaces syntax tree classes `Goto` and `Return`, default classes, with `MutexGoto` and `MutexReturn`, respectively.

```
class MutexGoto;
class MutexReturn;
void
MutexStmt::bind()
{
    lex->bindKeyword("goto", GOTO, MutexGoto);
    lex->bindKeyword("return", RETURN, MutexReturn);
}
```

The following program shows the code generation of the `mutex` statement.

```
MutexStmt::code()
{
    exitLabel = gensym();
    mutexp = gensym();
    if (arg1->getType() == base_typeid(Mutex)) {
        `{ Mutex  *$1; }.genCode(mutexp);
        `{ $1 = ($2).enter(); }.genCode(mutexp, arg1);
        body->genCode();
        `{ $1->leave(); }.genCode(mutexp);
        `{ $1: }.genCode(exitLabel);
    } else if (arg1->getType() == base_typeid(Mutex*)) {
        `{ Mutex  *$1; }.genCode(mutexp);
        `{ $1 = ($2)->enter(); }.genCode(mutexp, arg1);
        body->genCode();
        `{ $1->leave(); }.genCode(mutexp);
        `{ $1: }.genCode(exitLabel);
    } else {
        cerr << "illegal mutex operand in line " << this->lineno->no << endl;
    }
    return GEN_END;
}
```

We have to modify the code generation of statements `return` and `goto` appeared within the `mutex` statement. Before giving the code, it is better to describe the protocol that allows control statements `break`, `return`, and `goto` to be produced.

Let a *block statement* be a statement taking a compound statement. Statements `while` and `for` are examples of a block statement. Control statements `break`, `return`, and `goto` exit from nested block statements and change the context. The `break` statement exits from a block statement while the `return` statement exits from block statements and the function executed. To generate the correct code for exiting each block statement, the following protocol is defined: i) a subclass of `BlockStmt` represents a block statement, ii) `BlockStmt::curEnv` points to the block statement being generated, iii) `BlockStmt` has method `genLeave` which specifies the code generation for exiting the block statement. To generate the `break` statement code, method `genLeave` of an object pointed to by `BlockStmt::curEnv` is invoked. The code generation method of the `return` statement invokes all `genLeave` methods of block statements outside an object pointed to by `BlockStmt::curEnv`.

The `goto` statement exits from some nested block statements and may jump to another block statement. To ensure the correct code generation for jumping to a block statement, method `genEnter` will be defined in the block statement class.

The `mutext` statement involves the modification of control statements `return` and `goto`. We do not modify the `break` statement, but define `genEnter` and `genLeave` methods as follows.

```
Status MutexStmt::genEnter(Label *)
{
    cerr << "Cannot jump into the mutex statement\n";
    return ERROR;
}
Status MutexStmt::genLeave(Label *&lb)
{
    `{ ($1)->leave(); }.genCode(mutexp);
    lb = exitLabel;
    return OK;
}
```

We define classes `MutexReturn` and `MutexGoto` representing statements `return` and `goto`, respectively. The class id has been bound to the `return` keyword as shown in the `bind` method of `MutexStmt`.

```
class MutexReturn : public ReturnStmt {
public:
    virtual int         code();
};
MutexReturn::code()
{
    BlockStmt   *env = BlockStmt::curEnv;
    Label       *tmp;

    while (env) {
        env->genLeave(tmp);
        env = env->getOuter();
    }
    return GEN_CONT;
}
```

```
class MutexGoto : public GotoStmt {
public:
    virtual int         code();
};
MutexGoto::code()
{
    BlockStmt   *env = BlockStmt::curEnv;
    BlockStmt   *dstEnv = label->getEnv();
    Label       *tmp;

    while (env->outEnv(dstEnv)) {
        env->genLeave(tmp);
        env = env->getOuter();
    }
    `{ goto $1; }.genCode(label);
    return GEN_END;
}
```

In the above example, functions `blockBegin()` and `blockEnd()` are invoked to generate a compound statement. That is, by simply invoking `blockBegin()` followed by `blockEnd()`, the compound statement {} is generated.

5 Metaobject Protocol

Except for some new notations, the user can write a program at meta-level just as in C++ programming. The following syntax is introduced to declare a type, variable, or function, or to execute a statement in the meta-level:

meta-decl:
    `$meta` *decl-specifiers*$_{opt}$ *declarator-list*$_{opt}$ *';'*
    `$meta` *function-definition*
meta-exec:
    `$meta` *compound-statement*

`$meta` tells the compiler to shift to the meta-level. That is, all name table objects are used at the meta-level. For example, the following example declares global integer variable `g1` and the global `SynObj` pointer variable `sobj` that type `SynObj` is a predefined type in the meta-level.

```
$meta int       g1;
$meta SynObj    *sobj;
```

5.1 New Syntax

Meta-level declarator `$syntaxdef` allows the user to declare a new class derived from a predefined syntax tree class, and to register the keyword for the new modifier, specifier, expression, statement, or storage to the keyword table in the lexical analyzer object. Generally speaking, meta-lavel declarator `$syntaxdef` takes the following form:

`$syntaxdef` *syntax-type options keyword class-id* { *member-specification$_{opt}$* }

syntax-type is selected from `$modifier`, `$specifier`, `$struct`, `$expression`, and `$statement`. *keyword* is the identifier or the identifier followed by paired parentheses. *inst-vars-def* is the declaration of instance variables of the syntax object for the new syntax being registered with the meta-level declarator.

5.1.1 New Modifier

meta-modifier:
    `$syntaxdef $modifier` *mod-type identifier mod-arg$_{opt}$ class-id additional-mem-spec$_{opt}$* ';'
mod-type:
    `$decl` *or* `$spec` *or* `$member`
mod-arg:
    ( )
additional-mem-spec:
    { *member-specification$_{opt}$* }

Modifier `$decl` is used to register a declarator modifier, which means modifying the declarator '*'. Modifier `$spec` is used to register a specifier modifier, which means modifying the type specification or class member function. Modifier `$member` is used to register a member function modifier. The following are examples:

```
$syntaxdef $modifier $spec global GlobalModifier;
```

In the above meta-level declaration, the meta-level system behaves as if the following `GlobalModifier` class had been declared and method `init` had been invoked to register the keyword and class identifier pair in the lexical analyzer object.

```
class GlobalModifier : public SpecDecl {
private:
    virtual SynObj      *storage();
    virtual SynObj      *type();
public:
    static  void        init();
};
void GlobalModifier::init()
{
    lex->bindKeyword("global", SPECMODIFIER, GlobalModifier);
}
```

The following declaration is legal after the above meta-level declarator has been processed.

```
global int      *p;
```

The lexical analyzer object will create a new syntax object of class `GlobalModifier` where `global` represents the parse tree of the above declaration.

The following declaration is illegal because the global modifier modifies the pointer declaration.

```
int     global *p;
```

To make the above statement legal, the following statement must be declared.

```
$syntaxdef $modifier $decl global;
```

Programming designers often design the modified class declaration so that additional method invocation features are introduced. The following statement shows the guard expression `when` clause is introduced:

```
$syntaxdef $modifier $member when () WhenClause;
```

Then, the following declaration must be legal:

```
class A {
private:
        int     i;
public:
        foo(int j) when(i > j);
};
```

5.1.2 New Structure

meta-new-structure:
$syntaxdef `$struct` *identifier class-id additional-mem-spec*$_{opt}$ *';'*

Meta-level declarator `$syntaxdef` may create a new structural notation. The following is an example:

```
$syntaxdef $struct interface IntfStruct;
```

The following declaration is then llegal.

```
interface anObject {
    /* structural declaration */
};
```

5.1.3 New Specifier

meta-specifier:
`$syntaxdef $specifier` *identifier mod-arg*$_{opt}$ *class-id additional-mem-spec*$_{opt}$ *';'*

```
$syntaxdef $specifier entry() EntrySpec;
```

5.1.4 New Expression

meta-new-expresssion:
`$syntaxdef $expression $infix` *expr-id '('precedence ',' precedence ')'*
class-id additional-mem-spec$_{opt}$
`$syntaxdef $expression $prefix` *expr-id '(' precedence ')'*

class-id additional-mem-spec$_{opt}$

`$syntaxdef $expression $postfix` *expr-id '(' precedence ')'*

class-id additional-mem-spec$_{opt}$

expr-id:

string

precedence:

`newUnaryExpr` *or* `newPostfixExpr` *or* `newPrimaryExpr`

or `unaryExpr` *or* `postfixExpr` *or* `primaryExpr`

or `nameExpr` *or* `dimensionExpr`

Meta-level declarator `$syntaxdef` allows the programmer to introduce new expressions. In the following example, the new expression `''<-''` is introduced:

```
$syntaxdef $expression $infix "<-" (newPrimaryExpr, primaryExpr) SendExpr;

sub(tt t) { t <- 10; }
```

5.1.5 New Statement

meta-new-statement:

`$syntaxdef $statement` *statement-style identifier identifier*$_{opt}$ *class-id additional-mem-spec*$_{opt}$ *';'*

statement-style:

digit

`$label`

A new statement allowed in the MPC++ meta-level architecture is a keyword followed by a compound statement, a keyword with some arguments followed by a compound statement, or a new label statement followed by arguments. Meta-level declarator `$syntaxdef $statement` registers the keyword and its statement type in the lexical analyzer. The *digit* in the syntax declares how many arguments are required in the new statement. The following are examples:

```
$syntaxdef $statement 0 blkstm1 BlkStm1;
$syntaxdef $statement 1 blkstm2 BlkStm2;
$syntaxdef $statement 2 blkstm3 BlkStm3;
$syntaxdef $statement 1 blkstm41 blkstm42 BlkStm4;
sub ()
{
    blkstm1 { /* statement */ }                           // legal statement
    blkstm2 (/*expr*/) { /* statement */ }                // legal statement
    blkstm3 (/*expr*/; /*expr*/) { /* statement */ }      // legal statement
    blkstm41 (/*expr*/) { /* statement */ }               // legal statement
    blkstm41 (/*expr*/) {                                 // legal statement
        /* statement */
    } blkstm42 {
        /* statement */
    }
}

$syntaxdef $statement $label receive Receive;
sub2 ()
{
        int     i, j, k;

        receive entry(i, j):    ;                         // legal statement
        receive entry(i, j)[k]: ;                         // legal statement
}
```

5.1.6 New Storage Class and Extension

`$sytanxdef $storage` meta-level statement introduces new storage classes such as the `parallel` function. The syntax of the meta-level statement is as follows:

meta-new-storage:
`$syntaxdef $storage` *identifier class-id additional-mem-spec*$_{opt}$ *';'*

The following is an example:

```
$syntaxdef $storage parallel;
```

The following program is then legal.

```
parallel void pp() {    /* statement */    }
```

Meta-level statement `$syntaxdef $extstorage` introduces new types of memory storage such as the parallel variable.

meta-extension-storage:
`$syntaxdef $extstorage` *identifier class-id additional-mem-spec*$_{opt}$ *';'*

An example is shown below. Unlike a storage class, the keyword is placed after the variable name. Optionally, an expression may flow the keyword.

```
$syntaxdef $extstorage @ Parallel;
int     i@;
int     j@[10];
```

5.2 Meta-level Programming

5.2.1 Definition of SynObj Subclasses

Using the `$synobjdef` feature, the user can easily define a subclass of class `SynObj`.

meta-class-def:
  `$synobjdef` *context-spec*$_{opt}$ *class-id* `:` *class-id* { *member-specification*$_{opt}$ }
context-spec:
  `$unaryop (` *unary-operator* `,` *type-name-spec* `)`
  `$binaryop (` *binary-operator* `,` *type-name-spec* `,` *type-name-spec* `)`
type-name-spec:
  type-name
  ...

For example, the following `MutexGoto` class is defined as a subclass of `GotoStmt`, which is a subclass of `SynObj`.

```
$synobjdef MutexGoto : GotoStmt;
```

That is, the following declaration must be registered at the meta-level:

```
class MutexGoto : public GotoStmt {
public:
    virtual SynObj      *code();
};
```

5.3 Examples

We have developed several C++ extension examples. In the rest of this section, we define them in the C++ level by the MPC++ metaobject protocol (MOP).

5.3.1 Atomic Class

Using the MPC++ MOP protocol, the user can define the following program, which has the same semantics as the program in Figure 17:

```
$syntaxdef $modifier $spec atomic Atomic;
$syntaxdef $modifier $member guard() Guard;
```

The MPC++ MOP protocol can replace the program in Figure 18 with the following statements:

```
$synobjdef AtomicReturn : ReturnStmt;
$meta AtomicReturn::code() { `{ bb.leave(); }.genCode(); return GEN_CONT; };
```

The description of `Atomic` methods `declstruct` and `memberModify` is the same as that described in Section 4.6.1 except that keyword `$meta` is required.

```
$meta SynObj    *Atomic::declstruct()
{
        // same program as that in section 4.6.1
}
$meta SynObj    *Atomic::memberModify(Function *fp)
{
        // same program as that in section 4.6.1
}
```

5.3.2 Global Pointer

Class declarations in Figures 19, 20, and 21 can be replaced with the following MPC++ MOP statements:

```
$syntaxdef $modifier $decl global Global;
$synobjdef $unaryop("*", global*) GlobalPtrRefExpr : PtrRefExpr;
$synobjdef $binaryop("=", global*, ...) GlobalAsgnExpr : AsgnExpr;
```

Methods defined in Section 4.6.2 are replaced with the following program:

```
$meta SynObj* Global::storage()
{
    return `{ struct _globalStorage; }.genStorage();
}
$meta SynObj* Global::type()
{
    return `{ struct _globalStorage; }.getType();
}
```

```
$meta GlobalPtrRefExpr::code()
{
    SynObj      *dtype = next->getType();
    next->metaRep();
    `( _globalGet(&$1) ).genCode(next);
    return GEN_END;
}
$meta SynObj* GlobalPtrRefExpr::type()
{
    SynObj      *dtype = next->getType();
    return dtype->deRef(dtype);
}
```

```
$meta GlobalAsgnExpr::code()
{
    SynObj      *dtype = left->getType();
    left->metaRep();
    `( _globalAddrSet(&$1, (void*)($2)) ).genCode(left, right);
    return GEN_END;
}
$meta SynObj* GlobalAsgnExpr::type()
{
    SynObj      *dtype = left->getType();
    return dtype->deRef(dtype);
}
```

5.3.3 Persistent Object

The declaration of Class `Permanent` and its initialization in the lexical analyzer object shown in Figure 2.4.3 can be replaced with the following MPC++ meta-level statement:

```
$syntaxdef $modifier $spec permanent Permanent {
    void                genPOfunc(StructDecl *decl);
    void                genInitPO(Symbol *name, StructDecl *decl);
};
```

The methods' bodies of `genPOfunc` and `genInitPO` are the same as those described in Section 4.6.3 except that the `$meta` keyword is added before the method definition.

5.3.4 Remote Expression

The declaration of Classes `Parallel` and `RemoteExpr`, and their initialization in the lexical analyzer object shown in Figure 4.6.4 can be replaced with the following MPC++ meta-level statement:

```
$syntaxdef $extstorage "@" Parallel;
$syntaxdef $expression $infix "@" (postFixExpr, dimensionExpr)  RemoteExpr : BinaryExpr;
```

5.3.5 Mutex Statement

Classes `MutexStmt`, `MutexGoto`, and `MutexReturn` are declared using the MPC++ MOP as follows:

```
$syntaxdef $statement 1 mutex MutexStmt {
    Label       *exitLabel;
    SynObj      *mtexp;
};
$synobjdef MutexGoto : GotoStmt;
$synobjdef MutexReturn : GotoStmt;
```

`MutexStmt` methods `bind` and `code` are defined as follows:

```
$meta void MutexStmt::bind()
{
    keybind("goto", GOTO, MutexGoto);
    keybind("return", RETURN, MutexReturn);
}
```

```
$meta MutexGoto::code()
{
    BlockStmt   *env = BlockStmt::curEnv;
    BlockStmt   *dstEnv = label->getEnv();
    Label       *tmp;

    blockBegin();
    while (env->outEnv(dstEnv)) {
        env->genLeave(tmp);
        env = env->getOuter();
    }
    `{ goto $1; }.genCode(label);
    blockEnd();
    return GEN_END;
}
```

The following is the `MutexStmt`'s `code` method.

```
$meta MutexStmt::code()
{
    exitLabel = gensym();
    mutexp = gensym();
    if (arg1->getType() == base_typeid(Mutex)) {
        `{ Mutex  *$1; }.genCode(mutexp);
        `{ $1 = ($2).enter(); }.genCode(mutexp, arg1);
        body->genCode();
        `{ $1->leave(); }.genCode(mutexp);
        `{ $1: }.genCode(exitLabel);
    } else if (arg1->getType() == base_typeid(Mutex*)) {
        `{ Mutex  *$1; }.genCode(mutexp);
        `{ $1 = ($2)->enter(); }.genCode(mutexp, arg1);
        body->genCode();
        `{ $1->leave(); }.genCode(mutexp);
        `{ $1: }.genCode(exitLabel);
    } else {
        cerr << "illegal mutex operand in line " << this->lineno->no << endl;
    }
    return GEN_END;
}
```

6 Current Status of the MPC++ Meta-level Architecture

We are presently implementing the MPC++ compiler. The old version of the MPC++ meta-level architecture has been partly implemented[8]. Though the core idea is the same as in the old version, the description capability has been improved. This document does not cover the following features:

1. Template handling in the meta-level
   We are now considering how to handle templates.

2. Concrete definition of meta-level classes
   Some class declarations are briefly described in the appendix.

3. Other source transformation capabilities
   We believe that data parallel features such as data parallel C is also programmable using the MPC++ meta-level architecture. We intend to design a more useful library for program transformation.

Acknowledgment

The author would like to thank Professors Akinori Yonezawa, Satoshi Matsuoka, Jiro Tanaka, Kokichi Futatsugi, Takuo Watanabe, and Etsuya Shibayama, Dr. Mitsuhisa Sato, Dr. Yuji Ichisugi, and other members of the RWC massively parallel programming language working group for their crucial comments on the designing of the MPC++ meta-level architecture. Discussions on MPC++ with Atsushi Hori, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, and Jörg Nolte, members of the RWC Tsukuba Research Center, have been frequent and thorough. The author would like to thank members of the Tsukuba Research Center for their contributions.

Finally, the author would like to express his gratitude to Dr. Junichi Shimada, director of the RWC partnership.

References

[1] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Integrating Concurrency and Data Abstraction in the COOL Parallel Programming Language. Technical Report CSL-TR-92-511, Stanford University, 1992.

[2] K. Mani Chandy and Carl Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In G. Agha, P. Wegner, A. Yonezawa, editor, *Research Directions in Concurrent Object Oriented Programming*. MIT press, 1993.

[3] 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.

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

[5] Y. Gourhant and Marc Shapiro. FOG/C++: a Fragmented-Object Generator. In *1990 USENIX C++ Conference*, pp. pp.63–74, 1990.

[6] Andrew S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *COMPUTER*, Vol. 26, No. 5, pp. 39–51, 1993.

[7] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel. In *IMSA'92 International Workshop on Reflection and Meta-Level Architecutre*, pp. 23–35, November 1992.

[8] Yutaka Ishikawa. MPC++:Massively Parallel, Message Passing, Meta-Level Processing C++. POOMA'94, December 1994. The silde is obtained via http://www.rwcp.or.jp/people/ishikawa/mpc++.html.

[9] Yutaka Ishikawa. The MPC++ Programming Language V1.0 Specification with Commentary – Document Version 0.1 –. Technical Report TR-94014, Real World Computing Partnership, June 1994.

[10] Yutaka Ishikawa and Hideaki Okamura. A New Reflective Architecture: AL-1 Approach. In *The Second OOPSLA 91 Workshop on Reflection and Metalevel Architecures*, 1991.

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

[12] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA'93 Proceedings*, pp. 91–108, 1993.

[13] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

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

[15] A. Yonezawa S. Matsuoka, T. Watanabe. Hibrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *Fifth ECOOP*, July 1991.

[16] 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.

[17] R. Seliger. Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling, and Garbage Collection. In *1990 USENIX C++ Conference*, pp. 241–264, 1990.

[18] B. C. Smith. Reflection and Semantics in Lisp. In *Proceedings of 11th ACM Symposium on Principles of Programming Languages*, pp. 23–35, 1984.

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

A New Syntax Patterns

A.1 Specifier/Declarator

new-class-key:
    NEW_STRUCT
new-storage-class-specifier:
    NEW_STORAGE_CLASS
new-type-specifier:
    NEW_SPEC
new-decl-specifier:
    NEW_SPEC_MODIFIER *mod-arg*$_{opt}$
new-ptr-operator:
    ''* NEW_DECL_MODIFIER *mod-arg*$_{opt}$
    '&' NEW_DECL_MODIFIER *mod-arg*$_{opt}$
new-member-declaration:
    decl-specifier-seq$_{opt}$ *declarator* NEW_MEMBER_MODIFIER *mod-arg*$_{opt}$ *function-body*
    decl-specifier-seq$_{opt}$ *declarator* NEW_MEMBER_MODIFIER *mod-arg*$_{opt}$ *';'*
new-direct-declarator:
    declarator-id NEW_STORAGE_MODIFIER *mod-arg*$_{opt}$
mod-arg:
    (*argument-declaration-list*)
    dimension-declaration
dimension-declaration:
    dimension-declaration [*constant-expression*]

A.2 Expression

new-unary-expression:
    new-unary-expression NEW_EXPR_NEWUNARY_UNARY *unary-expression*
new-unary-expression:
    unary-expression NEW_EXPR_UNARY_DIMENSION *dimension-expression*
new-unary-expression:
    unary-expression NEW_EXPR_UNARY_NAME *name*
new-postfix-expression:
    new-postfix-expression NEW_EXPR_NEWPOSTFIX_POSTFIX *postfix-expression*
new-postfix-expression:
    postfix-expression NEW_EXPR_POSTFIX_PRIMARY *primary-expression*
new-postfix-expression:
    postfix-expression NEW_EXPR_POSTFIX_DIMENSION *dimension-expression*
new-postfix-expression:
    postfix-expression NEW_EXPR_POSTFIX_NAME *name*
new-primary-expression:
    new-primary-expression NEW_EXPR_NEWPRIMARY_PRIMARY *primary-expression*

new-primary-expression:
  primary-expression `NEW_EXPR_PRIMARY_DIMENSION` *dimension-expression*
new-primary-expression:
  primary-expression `NEW_EXPR_PRIMARY_NAME` *name*
dimension-expression:
  dimension-expression `[` $expression_{opt}$ `]`
  dimension-expression `(` $expression_{opt}$ `)`

A.3 Statement

statement:
  `NEW_STMT0` *compound-statement* $optional\text{-}compound\text{-}statement_{opt}$
  `NEW_STMT1` `(` $expression_{opt}$ `)` *compound-statement* $optional\text{-}compound\text{-}statement_{opt}$
  `NEW_STMT2` `(`$expression_{opt}$ `;` $expression_{opt}$ `)` *compound-statement*
    $optional\text{-}compound\text{-}statement_{opt}$
  `NEW_STMT3` `(` $expression_{opt}$ `;` $expression_{opt}$ `;` $expression_{opt}$ `)` *compound-statement*
    $optional\text{-}compound\text{-}statement_{opt}$
  `NEW_STMT4` `(` $expression_{opt}$ `;` $expression_{opt}$ `;`
    $expression_{opt}$ `;` $expression_{opt}$ `)` *compound-statement*
    $optional\text{-}compound\text{-}statement_{opt}$
  `NEW_STMT5` `(` $expression_{opt}$ `;` $expression_{opt}$ `;`
    $expression_{opt}$ `;` $expression_{opt}$ `;` $expression_{opt}$ `)` *compound-statement*
    $optional\text{-}compound\text{-}statement_{opt}$
  `NEW_STMT6` `(` $expression_{opt}$ `;` $expression_{opt}$ `;`
    $expression_{opt}$ `;` $expression_{opt}$ `;`
    $expression_{opt}$ `;` $expression_{opt}$ `)` *compound-statement*
    $optional\text{-}compound\text{-}statement_{opt}$
optional-compound-statement:
  `NEW_STMTOPT` *compound-statement*

B Syntax Tree Classes in Abstract Compiler

`This section is still being evolved.`

All syntax tree classes are derived from class `SynObj` declared as follows:

```
class SynObj {
private:
    class Lineno        *lineno;
    class NameTable     *nt;
    class Statement     *upper;
public:
    virtual void        parseEnter() { return cntxtEnter(); }
    virtual void        parseLeave() { return cntxtLeave(); }
    virtual void        cntxtEnter();
    virtual void        cntxtLeave();
    virtual SynObj      *genCode() { return code(); }
    virtual SynObj      *code();
};
```

B.1 Specification and Declarator

B.1.1 SpecDecl

```
class SpecDecl : public SynObj {
private:
    SpecDecl            *next;
    int                 repflag;
private:
    virtual SynObj      *storage();
    virtual SynObj      *type();
public:
    virtual SynObj      *genStorage() { return this->storage(); }
    virtual SynObj      *getType() { repflag ? return this : return this->type(); }
    virtusl SynObj      *declStruct() { return NIL; }
    virtual SynObj      *memberModify(Function *fp) { return fp; }
};
```

B.1.2 Storage Class Specifier

```
typedef enum {
    SC_TYPEDEF,         SC_EXTERN,      SC_STATIC,
    SC_AUTO,            SC_REGISTER
} ScType;
class ScSpec : public SpecDecl {
private:
    ScType              sctype;
private:
    virtual SynObj      *storage();
    virtual SynObj      *type() { return next->type(); }
};
```

B.1.3 Function Specifier

```
typedef enum {
    FCT_INLINE,         FCT_VIRTUAL
} FctType;
class FctSpec : public SpecDecl {
private:
    FctType             fcttype;
private:
    virtual SynObj      *storage();
    virtual SynObj      *type() { return next->type(); }
};
```

B.1.4 Qualifier Specifier

```
typedef enum {
    QF_CONST,           QF_VOLATILE
} QfType;
class QfSpec : public SpecDecl {
private:
    QfType              qftype;
private:
    virtual SynObj      *storage();
    virtual SynObj      *type() { return next->type(); }
};
```

B.1.5 Basic Type Specifier

```
typedef enum {
    TYPE_CHAR,          TYPE_SHORT,            TYPE_INT,              TYPE_LONG,
    TYPE_SINGED,        TYPE_UNSIGNED,         TYPE_FLOAT,            TYPE_DOUBLE,
    TYPE_SIGNED_CHAR,   TYPE_SIGNED_SHORT,     TYPE_SIGNED_INT,       TYPE_SIGNED_LONG,
    TYPE_LONG_DOUBLE,   TYPE_VOID,             TYPE_STRUCT,           TYPE_UNION,
    TYPE_USERTYPE,      TYPE_LONGLONG,         TYPE_ULONGLONG
} DataType;
class BasicType : public SpecDecl {
private:
    DataType            dt;
private:
    virtual SynObj      *storage();
    virtual SynObj      *type() { return this; }
};
```

B.1.6 Structure/Class Declarator

```
class StructDecl : public SpecDecl {
private:
    BaseSpec            *bases;
    AccDecl             *member;
private:
    virtual SynObj      *storage();
    virtual SynObj      *type() { return this; }
    virtual SynObj      *addPrivateMember(SynObj*);
    virtual SynObj      *addPublicMember(SynObj*);
    virtual SynObj      *addProtectedMember(SynObj*);
    virtual SynObj      *getAccDecl() { return member; }
    virtual SynObj      *getBaseSpec() { return bases; }
public:
    virtual SynObj      *declStruct() { return this; }
};
```

B.1.7 Union Declarator

```
class UnionDecl : public SpecDecl {
private:
    AccDecl             *member;
private:
    virtual SynObj      *storage();
    virtual SynObj      *type() { return this; }
};
```

B.1.8 BaseSpec

```
class BaseSpec : public SynObj {
private:
    StructDecl          *decl;
    BaseSpec            *next;
    Access              access;
    Vspec               vspec;
public:
    Base                *getDecl() { return decl; }
    BaseList            *getNext() { return next; }
};
```

B.1.9 Base

```
class Base : public SpecDecl {

};
```

B.1.10 AccDecl

```
class AccDecl : public SynObj {
private:
    DeclList            *decl;
    AccDecl             *next;
public:
    DeclList            *getDecl() { return decl; }
    AccDecl             *getNext() { return next; }
};
```

B.1.11 DeclComaList

```
class DeclComaList : public SynObj {
private:
    Storage             *decl;
    DeclComaList        *next;
public:
    Storage             *getDecl() { return decl; }
    DeclComaList        *getNext() { return next; }
};
```

An example represented by object `DeclComaList` is as follows:

```
struct {
    int         i;
    int         j;
} s1, s2;
```

B.1.12 Function Declaration

```
class FuncDecl : public SpecDecl {
private:
    int                 numargs;
    SpecDecl            *rettype;
    Symbol              *name;
    ParamList           *paramlist;
    SynObj              *cvlist;
    ModList             *modlist;
public:
    ParamList           *getParamList() { return paramlist; }
    SpecDecl            *searchModifier(const type_info&);
    virtual SynObj      *memberModify(Function *) { return this; }
};
```

B.1.13 Modifier List

```
class ModList : public SynObj {
private:
    SpecDecl             *decl;
    ModList              *next;
public:
};
```

B.1.14 Pointer Declarator

```
class PtrDecl : public SpecDecl {
private:
    virtual SynObj       *storage();
    virtual SynObj       *type() { return this; }
};
```

B.1.15 Reference Declarator

```
class RefDecl : public SpecDecl {
private:
    virtual SynObj       *storage();
    virtual SynObj       *type() { return this; }
};
```

B.1.16 New Specifier/Declarator Modifier

```
class NewSpecDeclModifier : public SpecDecl {
private:
    virtual SynObj       *storage() { return next->storage(); }
    virtual SynObj       *type() { return this; }
};
```

B.1.17 New Member Modifier

```
class NewMemModifier : public SpecDecl {
private:
    virtual SynObj       *storage() { return next->storage(); }
    virtual SynObj       *type() { return this; }
};
```

B.1.18 New Structure

```
class NewStructDecl : public StructDecl {

};
```

B.1.19 New Specifier

```
class NewSpec : public SpecDecl {
private:
    virtual SynObj       *storage() { return next->storage(); }
    virtual SynObj       *type() { return this; }
};
```

B.2 Storage

B.2.1 Storage

```
class Storage : public SynObj {

};
```

B.2.2 Function

```
class Function : public Storage {
private:
    FuncDecl            *decl;
    StructDecl          *belong;
    Stmt                *body;
public:
    Stmt                *getBody() { return body; }
    void                setBody(Stmt *newbody) { body = newbody; }
    FuncDecl            *getDecl() { return decl; }
    Stmt                *addStmt(SynObj*);
};
```

B.2.3 Variable

```
class Variable : public Storage {

};
```

B.2.4 Enum

```
class Enum : public Storage {

};
```

B.3 Statements

B.3.1 Statement

```
class Stmt : public SynObj {
public:
    void                blockBegin();
    void                blockEnd();
};
```

```
class StmtList : public Stmt {
public:
    Stmt                *stmt;
    StmtList            *next;
    Stmt                *getStmt();
    StmtList            *getNext();
    void                addStmt(Stmt*);
    StmtList(Stmt*);
};
```

```
class BlockStmt : public Stmt {
private:
    BlockStmt   *outer;
    NestLevel   *nl;
    void        envEnter(BlockStmt*);
    void        envLeave();
public:
    virtual SynObj      *genCode() { SynObj *ret;
                                     envEnter(this);
                                     this->code();
                                     envLeave();
                                     return NIL;
                                    }
    virtual BlockStmt   *getOuter();
    virtual void        parseEnter() { envEnter(this);
                                       lex->pushKeywordTable();
                                       this->bind(); }
    virtual void        parseLeave() { lex->popKeywordTable();
                                       envLeave(this); }
    virtual void        bind();
    virtual Status      genEnter();
    virtual Status      genLeave();
    void                blockBegin();
    void                blockEnd();
    static BlockStmt    *curEnv;
};
```

B.3.2 Declaration Statement

```
class DeclList : public Stmt {
private:
    SynObj              *decl;
    DeclList            *next;
public:
    SynObj              *getDecl() { return decl; }
    DeclList            *getNext() { return next; }
};
```

Member variable `decl` must points to an object derived from class `Storage` or class `DeclComaList`.

B.3.3 Expr Statement

```
class ExprStmt : public Stmt {
private:
    Expr                *expr;
    virtual void        code();
public:
    ExprStmt(Expr arg);
};
```

B.3.4 Compound Statement

```
class CompStmt : public BlockStmt {
private:
    SynObj              *body;
    virtual void        code();
public:
    CompStmt(SynObj *arg);
};
```

B.3.5 Label Statement

```
class LabelStmt : public Stmt {
private:
    BlockStmt           *belong;
    Label               *label;
    virtual void        code();
public:
    virtual BlockStmt   *getEnv();
    LabelStmt(Label *);
};
```

B.3.6 If Statement

```
class IfStmt : public Stmt {
private:
    Expr                *cond;
    Stmt                *thenPart;
    Stmt                *elsePart;
    virtual void        code();
public:
    void                setCond(Expr*);
    void                setThenPart(Stmt*);
    void                setElsePart(Stmt*);
    IfStmt(Expr *ex, SynObj *tp, SynObj *ep);
};
```

B.3.7 Switch Statement

```
class SwitchStmt : public BlockStmt {
private:
    Expr                *expr;
    CompStmt            *body;
    virtual void        code();
public:
    SwitchStmt(Expr *ex, SynObj *bd);
};
```

B.3.8 Case Statement

```
class CaseStmt : public Stmt {
private:
    Expr                *expr;
    virtual void        code();
public:
    CaseStmt(Expr *ex);
};
```

B.3.9 For Statement

```
class ForStmt : public BlockStmt {
private:
    Expr                *init;
    Expr                *cond;
    Expr                *inc;
    Stmt                *body;
    virtual void        code();
public:
    ForStmt(Expr *ex1, Expr *ex2, Expr *ex3; SynObj *bd);
};
```

B.3.10 While Statement

```
class WhileStmt : public BlockStmt {
private:
    Expr                *cond;
    Stmt                *body;
    virtual void        code();
public:
    WhileStmt(Expr *ex, SynObj *bd);
};
```

B.3.11 Do While Statement

```
class DoWhileStmt : public BlockStmt {
private:
    Expr                *cond;
    Stmt                *body;
    virtual void        code();
public:
    DoWhileStmt(Expr *ex, SynObj *bd);
};
```

B.3.12 Goto Statement

```
class GotoStmt : public Stmt {
private:
    Label               *label;
    virtual void        code();
public:
    GotoStmt(Label *lb);
};
```

B.3.13 Break Statement

```
class BreakStmt : public Stmt {
private:
    virtual void        code();
public:
    BreakStmt(1);
};
```

B.3.14 Return Statement

```
class ReturnStmt : public Stmt {
private:
    Expr                *expr;
    virtual void        code();
public:
    ReturnStmt(Expr *ex);
};
```

B.3.15 Continue Statement

```
class ContinueStmt : public Stmt {
private:
    virtual void        code();
public:
    ContinueStmt();
};
```

B.3.16 Extendable Statements

```
class NewStmt1 : public BlockStmt {
private:
    Expr                *arg1;
    Stmt                *body;
public:
    NewStmt1(SynObj *arg, NewStmt *bd);
};
```

```
class NewStmt2 : public BlockStmt {
private:
    Expr                *arg1;
    Expr                *arg2;
    Stmt                *body;
public:
    NewStmt2(Expr *a1, Expr *a2, Stmt *bd);
};
```

```
class NewStmt3 : public BlockStmt {
private:
    Expr                *arg1;
    Expr                *arg2;
    Expr                *arg3;
    Stmt                *body;
public:
    NewStmt3(Expr *a1, Expr *a2, Expr *a3, Stmt *bd);
};
```

```
class NewStmt4 : public BlockStmt {
private:
    Expr                *arg1;
    Expr                *arg2;
    Expr                *arg3;
    Expr                *arg4;
    Stmt                *body;
public:
    NewStmt4(Expr *a1, Expr *a2, Expr *a3, Expr *a4, Stmt *bd);
};
```

```
class NewStmt5 : public BlockStmt {
private:
    Expr                *arg1;
    Expr                *arg2;
    Expr                *arg3;
    Expr                *arg4;
    Expr                *arg5;
    Stmt                *body;
public:
    NewStmt5(Expr *a1, Expr *a2, Expr *a3, Expr *a4, Expr *a5, Stmt *bd);
};
```

```
class NewStmt6 : public BlockStmt {
private:
    Expr                *arg1;
    Expr                *arg2;
    Expr                *arg3;
    Expr                *arg4;
    Expr                *arg5;
    Expr                *arg6;
    Stmt                *body;
public:
    NewStmt6(Expr *a1, Expr *a2, Expr *a3, Expr *a4, Expr *a5, Expr *a6, Stmt *bd);
};
```

```
class NewLabelStmt : public LabelStmt {
private:
    BlockStmt           *belong;
    Label               *label;
    virtual void        code();
public:
    virtual BlockStmt   *getEnv();
    NewLabelStmt(Label *);
};
```

B.4 Expressions

B.4.1 Expression

```
class Expr : public SynObj {
private:
    Expr                *next;
};
```

B.4.2 Variable Name Expression

```
class VarExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    VarExpr(Expr *arg);
};
```

B.4.3 Coma Expression

```
class ComaExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    ComaExpr(Expr *arg);
};
```

B.4.4 Constant Expression

```
class ConstExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    ConstExpr(Expr *arg);
};
```

B.4.5 This Expression

```
class ThisExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    ThisExpr(Expr *arg);
};
```

B.4.6 Function Invocation Expression

```
class FuncCallExpr : public Expr {
private:
    int                 numargs;
    Expr                *pfixExpr;
    CallParam           *param;
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    Expr                *getFunc() { return pfixExpr; }
    CallParam           *getParam() { return param; }
    FuncCallExpr(Symbol *fname, CallParam *cp);
    FuncCallExpr(Expr *exf, CallParam *cp);
};
```

B.4.7 Function Invocation Parameter

```
class CallParam : public Expr {
private:
    Expr                *expr;
    CallParam           *next;
private:
    virtual void        code();
};
```

B.4.8 Dot Expression

```
class DotExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    DotExpr(Expr *arg);
};
```

B.4.9 Pointer Expression

```
class PtrExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    PtrExpr(Expr *arg);
};
```

B.4.10 Post/Pre Inc/Dec Expressions

```
class PostIncExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    PostIncExpr(Expr *arg);
};
```

```
class PostDecExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    PostDecExpr(Expr *arg);
};
```

```
class PreIncExpr : public Expr {
private:
    virtual void       code();
    virtual SynObj     *type();
public:
    PreIncExpr(Expr *arg);
};
```

```
class PreDecExpr : public Expr {
private:
    virtual void       code();
    virtual SynObj     *type();
public:
    PreDecExpr(Expr *arg);
};
```

B.4.11 Assignment Expression

```
class AsgnExpr : public Expr {
private:
    virtual void       code();
    virtual SynObj     *type();
public:
    AsgnExpr(Expr *arg);
};
```

B.4.12 Conditional Expression

```
class CondExpr : public Expr {
private:
    virtual void       code();
    virtual SynObj     *type();
public:
    CondExpr(Expr *arg);
};
```

B.4.13 Unary Logical Expression

```
class UnaryExpr : public Expr {
private:
    virtual void       code();
    virtual SynObj     *type();
public:
    UnaryExpr(Expr *arg);
};
```

B.4.14 Binary Logical/Arithmetic Expression

```
class BinaryExpr : public Expr {
private:
    virtual void       code();
    virtual SynObj     *type();
public:
    BinaryExpr(Expr *arg);
    BinaryExpr(BinaryOp, Expr*, Expr*);
};
```

B.4.15 Sizeof Expression

```
class SizeOfExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    SizeOfExpr(Expr *arg);
};
```

B.4.16 Cast Expression

```
class CastExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    CastExpr(Expr *arg);
};
```

B.4.17 New Expression

```
class NewExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    NewExpr(Expr *arg);
};
```

B.4.18 Delete Expression

```
class DeleteExpr : public Expr {
private:
    virtual void        code();
    virtual SynObj      *type();
public:
    DeleteExpr(Expr *arg);
};
```

B.4.19 New Unary Expression

```
class NewUnaryExpr : public Expr {
};
```

B.4.20 New Binary Expression

```
class NewBinaryExpr : public Expr {
};
```

B.4.21 New Trinary Expression

```
class NewTrinaryExpr : public Expr {
};
```