January 1997. To appear in Nordic Journal of Computing.
Special Issue on Programming Environments.

Attribute Extension — A Technique for Enforcing Programming Conventions

Görel Hedin

Dept of Computer Science, Lund Institute of Technology
Box 118, S-221 00 Lund, Sweden
e-mail: Gorel.Hedin@dna.lth.se

Abstract. A problem in supporting reusability of software libraries and frameworks is that the programming conventions which need to be followed are only informally described. Safer reuse would result if these conventions could be enforced, preferably at compile time. This paper presents a technique supporting such enforcement. The technique is based on attribute grammars and allows the construction of extensible compilers and checkers.

Keywords: domain-specific languages, extensible languages, reusability, class libraries, object-oriented frameworks, attribute grammars, extensible compilers.

1 Introduction

Libraries as language extensions

One of the most important ways to obtain reuse in software development is to construct reusable libraries of program components. A library can be thought of as a domain-specific language, because abstraction mechanisms such as classes and methods allow the modelling of domain-specific concepts. In object-oriented frameworks this idea is often taken even further by embedding also the main application behavior in the library, thus making the library more of an extensible application than an application subcomponent.

User interface frameworks such as MacApp and the Smalltalk Model-View-Controller (MVC) made OO frameworks widespread and recognized as a powerful reuse technique [Deu89, JF88], but the idea of OO frameworks dates back to Simula with class Simulation which provided an object-oriented framework for discrete event simulation. The view of OO programming as a language-extension technique was explicitly stated as one of the main points in Simula, where one can read the following in the preface of the language definition [DMN68]:

> *"A main characteristic of SIMULA is that it is easily structured towards specialized problem areas, and hence can be used as a basis for Special Application Languages."*

In the following, we will use the term *library* to cover ordinary procedural libraries, class libraries, and object-oriented frameworks. A problem with reusing libraries is that the programming conventions which need to be followed in order for the library to work properly, are only informally described (if they are described at all). Robust libraries may include run-time checking of some of the conventions, but it is usually not possible to add run-time checks for all possible convention violations. The result is that convention violations lead to run-time errors, often occurring deep within the libraries, and which are very difficult for the application programmer to understand and debug.

Library-specific static-semantic checks

The goal of this paper is to pursue the idea of a library as a language even further by allowing *library-specific static-semantic checking* to be added to a base language. We suggest a way of specifying such checks and of how to construct an extensible compiler which can be extended to perform these checks. Our technique allows more errors to be caught, errors are caught earlier (at compile-time rather than at run-time), and they are associated to relevant points in the program.

We have chosen to allow extension only of static-semantic checks, but not extension of the context-free syntax, or changing the static- or dynamic semantics of the base language. Thus, any program accepted by the extended compiler will also be accepted by an ordinary base language compiler and will produce the same dynamic behavior. This has several advantages. In particular, the application programmer can take full advantage of production-quality base language compilers and other tools (e.g. debuggers), and does not need to learn any new syntax. To also allow extension of base language syntax and semantics could make it easier to specify the library-specific checks, and could make application programming more convenient. However, we believe there is a large set of problems where the advantages of keeping compatible with the base language outweighs the advantages of defining domain-specific syntax.

The syntax and semantics of the base language has a paramount influence on how difficult it is to specify the library-specific checks. Our approach can be applied to any programming language based on a context-free grammar. However, it is primarily intended for base languages in the statically typed object-oriented family, such as C++, Java, Eiffel, Simula, or BETA. The static typing and the data abstraction and specialization mechanisms of these languages provides a suitable basic semantic framework necessary for being able to easily add many kinds of library-specific checks.

In this paper, we focus on enforcing library-specific conventions. However, the technique for adding user-specified semantic checks is also appropriate for enforcing general programming style conventions, or programming "laws" such as the "Law of Demeter" [LH89], and similar restrictions on how the base language should be used.

Other approaches to enforcing programming conventions are Minsky's Law-Governed Architecture [Min95, MiP94] and Meyer's CCEL system [MDR93, CM93b]. In Section 6 we will compare these approaches to ours.

Attribute extension

To enforce programming conventions we propose a new technique, *attribute extension*, which allows a static-semantic checker for a base language to be extended with library-specific checks. The checker makes name bindings, types, and other static-semantic properties of the base language available through a *base grammar interface*, specified in an object oriented form. An *extension grammar* can be written for a given library, specifying what checks to perform on application programs using that library. The extension grammars are a kind of attribute grammars which extend the base grammar interface.

Paper organization

The rest of this paper is organized as follows. In Section 2 we give an example of a monitor library, illustrating how the choice of base language can affect the number of programming conventions which need to be enforced. In Section 3 we describe the attribute extension mechanism. Section 4 gives an example of an extension grammar. In Section 5 we discuss implementation considerations. Section 6 relates our approach to other work and Section 7 summarizes the results.

2 Language impact on programming conventions

The nature and level of the base language naturally has a great impact on what programming conventions are needed for a given problem, and how difficult they are to express. We will illustrate this by investigating a series of languages with increasing abstraction capabilities, ranging from only procedural abstraction to a language with abstraction and specialization mechanisms for both procedures and data. The example we will investigate is a library for concurrent programming which includes an emulation of the monitor construct.

A *monitor* [Hoa74, Bri75] is a language construct similar to a class or abstract data type where special *entry procedures* guarantee mutually exclusive access to the encapsulated data. Furthermore, it is possible to associate a *condition* with an entry procedure in order to delay processes calling the entry procedure until the condition is fulfilled, e.g. delaying a call to a procedure `get` on a buffer until there is at least one element in the buffer.

To construct the monitor library component, we assume there is already a library component for semaphores available. It is then straight-forward to emulate the dynamic behavior of monitors, as explained in standard textbooks on concurrent programming. We will investigate what programming conventions an application programmer needs to follow to make the monitor library work properly, and how the programming conventions differ depending on the base language chosen.

We will look at three categories of base languages. The first base language is a procedural language, like C or Pascal, which contains procedure and record type constructs. The second base language is an object-oriented language, like C++, Java, or Simula, which contains a data abstraction facility (the class) and a data specialization mechanism (subclassing). The third base language is an object-oriented language which also includes a mechanism for specialization of procedures, like BETA [MMN93]. Libraries for concurrent programming, containing emulated monitor constructs, exist for all of these language categories. (Of course, Java does not need such a library since it contains a built-in language construct for monitors.) At Lund Institute of Technology, such libraries have been used for many years based on Pascal, Modula-II, and recently on Simula [Mag96]. Emulation of monitors in BETA is described in [MMN93].

2.1 A procedural base language

Figure 1 shows the interface of a library which can be used to program emulated monitors in a procedural language, using Pascal syntax. The library contains a pointer type MonitorVariable (pointing to a record containing the semaphores needed for synchronization of the monitor), and procedures initMonitor, enterMonitor, exitMonitor, and awaitMonitorChange.

```
type MonitorVariable = ^MonitorVariableRecord; (* contains semaphores for the monitor *)
procedure initMonitor (M: MonitorVariable) ...
   (* Initializes the monitor variable *)
procedure enterMonitor (M: MonitorVariable) ...
   (* Waits until the monitor is free, then enters it (locks it) *)
procedure exitMonitor (M: MonitorVariable) ...
   (* Leaves the monitor (unlocks it). Also causes processes awaiting change to be put on
   queue for entering the monitor. *)
procedure awaitMonitorChange (M: MonitorVariable) ...
   (* Leaves the monitor and waits until some other process has visited the monitor. Then
   enters the monitor again. *)
```

Figure 1 *Library: Procedural interface for building monitors*

When using this library to implement a specific monitor, the conventions listed in Figure 3 should be followed. Figure 2 outlines an example application program with a FIFO bounded buffer monitor which follows these conventions.

Even if we assume that the application programmer is cooperative and *tries* to follow the conventions, there are several mistakes which are easy to do and which lead to severe errors. For example:

- Omission of a call to enterMonitor may lead to unsynchronized updates of the buffer, leading to inconsistencies in the buffer representation and to run-time errors (violation of convention 3.)
- Access to monitor data outside of an entry procedure gives similar errors (violation of convention 4).
- Omission of a call to exitMonitor may lead to deadlock (violation of convention 3).

An additional problem is that it is possible to use the library without following the conventions, and still get a program which works, but which is very difficult to read and maintain. Examples include placing the monitor variable outside the record containing the protected data, or placing the enterMonitor and exitMonitor calls at the entry procedure call sites rather than inside the entry procedures. These are common errors made by students. The conventions thus promote both a uniform and correct way of using the library.

```
type
    (* monitor *) FIFOMonitor = ^ FIFOMonitorRecord;
    FIFOMonitorRecord = record
        M: MonitorVariable;
        BB: BoundedBuffer; (* the protected data *)
    end;

(* init *) procedure InitFIFO (B: FIFOMonitor);
begin
    new (B^.M);
    initMonitor(B^.M);
    new (B^.BB);
    initBuffer(B^.BB);
end;

(* entry *) procedure put (B: FIFOMonitor; E: Element);
begin
    enterMonitor (B^.M);
    (* condition *) while boundedBufferFull(B^.BB) do awaitMonitorChange(B^.M);
    boundedBufferAddAsLast(B^.BB, E);
    exitMonitor (B^.M);
end;

(* entry *) function get (B: FIFOMonitor) : Element;
begin
    enterMonitor (B^.M);
    (* condition *) while boundedBufferEmpty(B^.BB) do awaitMonitorChange(B^.M);
    get := boundedBufferRemoveFirst(B^.BB);
    exitMonitor (B^.M);
end;
```

Figure 2 *Application: Procedural implementation of a FIFO monitor*

2.2 An object-oriented base language

In an object-oriented language, data abstraction (classes) and data specialization (subclassing) can be used to emulate monitors with much simpler programming conventions than in the procedural case.

The primitive monitor operations can be encapsulated in an abstract class Monitor, and the application program can define specialized monitors by creating subclasses to Monitor. This provides a simpler interface and a much nicer application program structure than in the procedural case, as seen from Figures 4 and 5 (using a Java-like syntax). In particular, the existence of the monitor variable and the monitor initialization is completely hidden from the application programmer.

The object-oriented solution is much simpler than the procedural one. Note, however, that the application programmer is still required to follow the conventions of adding enter, exit and awaitChange calls at the appropriate places in the entry methods. Figure 6 describes the conventions needed to be followed by the application.

Note that for two of the conventions, no checks are needed at all: Monitor emulation (1) can only be done in one way, and there is therefore no need for any

1. **Monitor emulation**. The monitor should be emulated by a record type R containing a variable M of the pointer type MonitorVariable. Additional data D in R constitutes the data to be protected by the monitor.
2. **Entry procedure emulation**. An entry procedure of R should be emulated by a normal procedure which has a pointer to an R-instance as one of its parameters.
3. **Entry procedure implementation**. An entry procedure should have a call to enterMonitor(R.M) as its first statement and a call to exitMonitor(R.M) as its last statement. Calls to enterMonitor and exitMonitor are not allowed in other places in the program.
4. **Data protection**. The data D may only be inspected and changed by entry procedures of R, or by procedures called by the entry procedures.
5. **Conditional entry emulation**. Conditional entry of the monitor should be programmed by a statement

   while *condition* do awaitMonitorChange(M);

   as the second statement in the entry procedure. Calls to awaitMonitorChange must not occur in other places in the program.
6. **Initialization**. For each monitor instance (instance of R), exactly one call to initMonitor(R.M) must be done. This call must be done before any calls to the entry procedures of R.

Figure 3 *Conventions for the procedural library*

checks. Initialization (6) is automatically taken care of by the library. For the other conventions, the checks are much simpler to express. This is because the realization of the monitor construct as a class gives a syntactic relation between the monitor variable, the primitive monitor operations, the entry methods, and the protected data. In the procedural solution, these relations are all based on parameters, making the conventions involved more complex.

(For simplicity, we assume that the **return** construct only sets the return value of a function. For Java and C semantics where the **return** construct also returns control to the caller, one would need slightly more complex programming conventions: entry methods would need to save the return value in a local variable v, and place the statement "**return** v" *after* the call to exit.)

```
public class Monitor {
    protected void enter() {...};
    protected void exit() {...};
    protected void awaitChange() {...};
};
```

Figure 4 *Library: Interface to an abstract monitor class*

```
public class FIFOMonitor extends Monitor {
    private BoundedBuffer BB = new BoundedBuffer();

    /* entry */ public void put (Element E) {
        enter();
        /* condition */ while (BB.full()) awaitChange();
        BB.addAsLast(E);
        exit();
    };

    /* entry */ public Element get () {
        enter();
        /* condition */ while (BB.empty()) awaitChange();
        return BB.removeFirst();
        exit();
    };
};
```

Figure 5 *Application: Object-oriented implementation of a FIFO monitor*

1. **Monitor emulation**. *No checks are needed*. A monitor is emulated by a subclass C to the abstract class Monitor. The data D in C constitutes the data to be protected by the monitor.
2. **Entry procedure emulation**. An entry procedure should be emulated by a method in C.
3. **Entry procedure implementation**. An entry method should have a call to enter as its first statement and a call to exit as its last statement. Calls to enter and exit are not allowed in other places in the program.
4. **Data protection**. The data D may only be inspected and changed by entry methods of C, or by methods called by the entry methods.
5. **Conditional entry emulation**. Conditional entry of the monitor should be programmed by a statement

   while *condition* do awaitChange;

   as the second statement in the entry method. Calls to awaitChange must not occur in other places in the program.
6. **Initialization**. *No checks are needed*. Initialization of internal monitor semaphores is done automatically at the creation of a Monitor instance.

Figure 6 *Conventions for the object-oriented library*

2.3 An object-oriented base language with submethoding

It is possible to obtain a library with even simpler programming conventions if the object oriented language supports *procedural specialization*, i.e. if a method can have submethods, in analogy to a class having subclasses. Such submethoding is available in the object-oriented language BETA [MMN93]. A *submethod* extends its supermethod in the following way: The supermethod may contain a statement inner which causes the code of the submethod to be executed. Submethods may declare additional input and output parameters (return values). The inner construct originates from Simula where it is used in class bodies. The idea of submethods was also proposed in [Vau75] where monitors was one of the principal examples used. The use of subclassing and submethoding for modeling monitors is also treated in [LM81] and [MMN93].

Figure 7 illustrates how submethoding is used to encapsulate the calls to enter and exit in an abstract method entry. In the application, (Figure 8) the entry procedures are defined as submethods to entry, and the calls to enter and exit are thereby automatically taken care of by the library. (We use an extension of Java syntax rather than BETA to make the syntax more familiar to most readers.)

```
public class Monitor {
    protected void entry() {
        boolean condition() { return false; };

        enter();
        while condition() do awaitChange();
        inner;
        exit();
    };
};
```

Figure 7 *Library: Interface with abstract method* entry

```
public class FIFOMonitor extends Monitor {
    private BoundedBuffer BB = new BoundedBuffer();

    public void put (Element E) extends entry {
        void condition () { return BB.full() };
        BB.addAsLast(E);
    };

    public Element get () extends entry {
        void condition () { return BB.empty() };
        return BB.removeFirst();
    };
};
```

Figure 8 *Application: Using submethoding*

Also the while-loop for conditional entry is encapsulated in the abstract entry method in the library, removing this burden from the application program. In order to specify the condition, a local method condition is defined in entry, and submethods of entry can override this local method to return the appropriate condition value. This technique relies on the unlimited block-structuring available in BETA, allowing classes and methods to have local classes and methods.

The use of inner in submethoding is somewhat similar to the Template Method design pattern [GHJV94]. A template method defines the skeleton of an algorithm, deferring some steps to subclasses by calling methods which are overridden by a subclass. In a similar way, the method entry is a skeleton where the actions taken at the point of inner are deferred to a subclass.

However, the Template Method design pattern is different from submethoding, and can not be used to solve our problem. The best we could do with the Template Method would be to define two template methods put and get in class Monitor, and let them call abstract methods putCondition and putAction, and getCondition and getAction, respectively. These abstract methods would then be implemented in subclasses to Monitor. However, this would restrict our monitor functionality to monitors with exactly two entry methods, and would furthermore make it necessary to decide on the number and types of parameters already in the library. In contrast, in a library based on submethoding, applications can define monitors with any number of entry methods and with parameters decided by the application.

Thus, a library based on Template Methods does not contain support for implementing general monitors, and is much too restrictive.

By using submethoding, the conventions that application programs need to adhere to are further simplified as shown in Figure 9. Here, checks are only needed for data protection (convention 4).

1. **Monitor emulation**. *No checks are needed*. A monitor is emulated by a subclass C to the abstract class Monitor. The data D in C constitutes the data to be protected by the monitor.
2. **Entry procedure emulation**. *No checks are needed*. An entry procedure is emulated by a submethod to entry.
3. **Entry procedure implementation**. *No checks are needed*. Calls to enter and exit are automatically taken care of by the library.
4. **Data protection**. The data D may only be inspected and changed by entry methods of C, or by methods called by the entry methods.
5. **Conditional entry emulation**. *No checks are needed*. Conditional entry of the monitor is programmed by overriding the local method condition in the entry method.
6. **Initialization**. *No checks are needed*. Initialization of internal monitor semaphores is done automatically at the creation of a Monitor instance.

Figure 9 *Conventions for the object-oriented library based on submethoding*

2.4 Summary

We have seen in this example how the choice of base language dramatically influences the complexity of the programming conventions which need to be followed by an application programmer. However, even a very advanced language like BETA can not capture all programming conventions. Library usage can still be made safer by adding library-specific checks. We have also seen that by using an object-oriented language, the structure imposed by the class abstractions makes it easier to express the programming conventions.

3 Attribute extension

To do library-specific checking, a checker knowledgeable of the library-specific rules should be run on each application program using the library. Implementing such a checker from scratch for each library would be a large task, since it would need to redo much of the name and type analysis for the base language in order to proceed with the library-specific checks.

Instead, it is desirable to have an *extensible checker* which performs name- and type analysis for the base language, and to which library-specific checks can be easily added. When running the extended checker on a given application program, the checks for each of the used libraries are performed.

We propose a technique called *attribute extension* to build such an extended checker. This technique makes use of the following specifications:

- A *base grammar interface*, describing name binding information, type information, and other useful general information about a program in the base language.
- An *extension grammar*, describing the library-specific static-semantic checks, making use of the base grammar interface.
- *Attribute comments*, annotating an application program to describe relations to the library that cannot be expressed in the base language.

(With "application program" we mean any source code element using the library. An application program could itself be a library rather than a complete program.)

3.1 Base grammar interface

The base grammar interface is a context-free grammar extended with function signatures. These function signatures are similar to attribute declarations in an attribute grammar, but are more general in that:

1. A function may take one or more arguments, whereas an attribute corresponds to a function without arguments.
2. The function arguments and result values may be references to syntax tree nodes or other objects. In contrast, attributes in an attribute grammar are usually only allowed to have applicative values (they cannot be references to objects).

An example base grammar interface

The context-free grammar of the base language is specified using an object-oriented grammar notation [Hed89]. Both nonterminals and productions are mapped to *node classes*, which are arranged in an inheritance hierarchy. Typically, nonterminals correspond to superclasses whereas productions correspond to subclasses. It is also possible to have more than two levels in the node class hierarchy, to use abstract superclasses to model aspects on a more general level than nonterminals and productions. In the example grammar below we make use of three such abstract superclasses: Node - the abstraction of any syntax node in a base language program, Descendant - the abstraction of any node except for the root node, and Root - the abstraction of the root node. These node classes are also convenient when specifying general behavior in the extension grammars.

Figure 10 shows parts of a base grammar interface for an object-oriented language. It includes node classes for some aspects of class declarations, method declarations, and statements.

```
nodeclass Node ::= {
    integer function cardinal()
}
nodeclass Descendant extends Node ::= {
    integer function childNo()
}
nodeclass Root extends Node ::= { }

nodeclass Declaration extends Descendant ::= {
    string function globalName()
    boolean function public()
    boolean function protected()
    boolean function private()
}
nodeclass ClassDecl extends Declaration ::=
    ( optional SuperClassId, ClassId, DeclList) {
    ClassDecl function superClassBinding()
}
nodeclass DeclList extends Descendant ::= list of Declaration { }
nodeclass VarDecl extends Declaration ::=
    (Type, VarId, optional Expression) {
}
nodeclass MethodDecl extends Declaration ::=
    (optional ReturnType, MethodId, FormalParamList, MethodBody) {
}
nodeclass MethodBody extends Descendant ::= list of Statement { }
nodeclass Statement extends Descendant ::= { }
nodeclass MethodCall extends Statement ::=
    (MethodId, ActualParamList) {
    MethodDecl function methodDeclBinding()
}
nodeclass WhileStmt extends Statement ::= (Expression, Statement) { }
```

Figure 10 *A base grammar interface (incomplete)*

The most general class Node has a function cardinal which returns the number of children. The children have sequence numbers from 1.. cardinal, and the function childNo in Descendant returns that number.

The node class Declaration has a function globalName which returns a globally unique name of the declaration, e.g. “L:C:m” for a method m in a class C in a library L. It also has functions public, protected, and private, which return *true* if the declaration is a method or variable declared with any of these modifiers.

These functions are inherited by ClassDecl, VarDecl, and MethodDecl which extend (are subclasses of) Declaration.

Constructs which access named entities, have a function returning a reference to the corresponding entity: MethodCall has a function methodDeclBinding which returns a reference to the corresponding method declaration. ClassDecl has a function superClassBinding which returns a reference to the declaration of the superclass.

3.2 Extension grammar

The extension grammar for a library describes the library-specific checks to be carried out on an application program using the library. The extension grammar is an attribute grammar extending the classes in the base grammar interface with *attribute declarations* and *equations*. An equation has the form

eq *Attr* = *Exp*

where *Attr* is an attribute and *Exp* is an expression over other attribute values. An attribute is declared as either *synthesized* (used to propagate information upwards in the syntax tree) or *inherited* (used to propagate information downwards). It is the responsibility of each node class C to declare equations defining the synthesized attributes in C and the inherited attributes in son nodes to C. This implies that there is exactly one defining equation for each attribute. Provided that there are no circular definitions, all attribute values will then be unambiguously defined.

As in a standard attribute grammar, the right-hand side expression *Exp* may access attributes in the syntax node itself or in the son nodes. In addition, *Exp* may call functions in the base grammar interface, and access attributes and functions in the (remote) syntax nodes returned by such functions. Thus, although most attribute dependencies in the extension grammar are local, it can use the base grammar interface to propagate non-local information, e.g. along decl-use bindings. This will be exemplified in Section 4.2.

Since the context-free grammar (defined in the base grammar interface) has the form of a class hierarchy, the usual benefits from object-oriented descriptions are obtained, i.e., the possibility to describe behavior at appropriate levels of abstraction and to override default behavior for specific cases. Attributes and equations are inherited (in the object-oriented sense) from superclasses to subclasses, and equations in subclasses can override equations in superclasses. This makes it easy to define general behavior applying to many node classes without having to clutter the grammar with many similar attribute declarations and equations, as is a normal problem with standard AGs. E.g., by placing equations and/or attribute declarations in the most general node class Node, the behavior will apply to all nodes in the syntax tree. Examples of this given in Section 4.

In addition to the usual synthesized and inherited attributes, an extension grammar includes *program-defined attributes* and *error attributes*, as discussed in the following sections. The error attributes are the ones actually describing the library-specific checks to be performed.

3.3 Attribute comments

Some relations between a library and an application program cannot be easily described using the base language itself. For example, the programming conventions for the object-oriented monitor library in Section 2.2 stipulates that there are two kinds of methods in a Monitor subclass: entry methods and non-entry methods. We propose that this distinction is made by annotating the application program with a specific kind of comments: If a comment

```
/*= entry =*/
```

is added in front of a method declaration, it will be regarded as an entry method. The application programmer can think of this comment as a modifier of the method construct.

We call these comments *attribute comments*. An attribute comment corresponds to an equation which overrides the default equation given in the extension grammar for a particular syntax node instance. Attributes whose definitions can be overridden like this are called *program-defined attributes*. An attribute comment has the following general form:

/*= *attribute* = *literal-value* =*/

where *literal-value* is a literal value of a primitive type, for example a literal boolean value (true or false), a literal numerical value (0, 1, 2, ...), or a literal string value. Because it is very common for program-defined attributes to be boolean, we allow the attribute comments for such attributes to be abbreviated to simply the name of the attribute, meaning that the attribute is defined to true. For example, /*= entry =*/ is equivalent to /*= entry = true =*/.

Program-defined attributes thus give an application program the possibility to control the attribute values of individual syntax node instances. The attribute comment /*= entry =*/ sets the entry attribute of a particular method to true. The default definition of the entry attribute is specified in the extension grammar as follows:

```
addto MethodDecl {
   progdef boolean entry = false;
};
```

The extension grammar makes use of the construct **addto** to indicate what node class in the base grammar the attribute declarations and equations belong to. In the above example, the (default) equation for entry is given directly in the declaration. This is shorthand for giving the equation separately as below:

```
addto MethodDecl {
   progdef boolean entry;
   eq entry = false;
};
```

The extension grammar says that the default value of entry is false for each method. In the application program, the entry attribute can be given the value true for individual methods as follows:

```
/*= entry =*/ put (Element E) ...
/*= entry =*/ Element get () ...
```

3.4 Error attributes

The output from running the extended checker for a library on an application program is a set of error messages attached to the language constructs in the application program, indicating faulty use of the library. To support this, the extension grammar declares some attributes as *error attributes*. An error attribute is a string-valued synthesized attribute which should be defined as an appropriate error string if there is a library-specific error, and the empty string otherwise.

For example, to check that an entry method contains a call to enter as its first statement we could add an error attribute missingEnter to the language construct MethodDecl as follows:

```
addto MethodDecl {
  error string missingEnter =
      if entry and not firstStatementIsEnter
      then "Missing call to enter"
      else "":
};
```

The definition of missingEnter makes use of a synthesized boolean attribute firstStatementIsEnter described below.

3.5 Access to specific named entities

In writing the specification of the library-specific checks, we need to be able to refer to particular named entities in the library. For example, in defining the attribute firstStatementIsEnter used above, we need to refer to the method enter to check if the first statement of an entry method is actually a call to enter.

Since many library-specific checks need to refer to named entities, the base grammar interface contains a function globalName which returns a unique name for each declaration in a program. For example, the global name for the enter method is MonitorLib:Monitor:enter, where MonitorLib is the name of the library containing the Monitor class.

Using the globalName function, we can define an attribute globalMethodName for all statements, defined as the global name of the method if it is a method call, and the empty string for all other statements. The attribute globalMethodName can be used to define an attribute globalMethodNameOfFirstStatement for a method body, and this attribute can in turn be used to define the attribute firstStatementIsEnter:

```
addto Statement {
   syn string globalMethodName = “”; default equation
}

addto MethodCall {
   eq globalMethodName = methodDeclBinding().globalName();
   overrides equation in superclass Statement
}

addto MethodBody {
   syn string globalMethodNameOfFirstStatement =
      if cardinal() = 0
      then “”
      else Statement[1].globalMethodName;
}

addto MethodDecl {
   syn boolean firstStatementIsEnter =
         (MethodBody.globalMethodNameOfFirstStatement =
“MonitorLib:Monitor:enter”);
}
```

The equation defining the attribute globalMethodName in MethodCall overrides the default definition given in Statement (the superclass of MethodCall). This equation calls the function methodDeclBinding available in MethodCall and then calls the function globalName on the result from methodDeclBinding.

The definition of the attribute globalMethodNameOfFirstStatement shows the use of list node classes: MethodBody is a list node class with a sequence of Statement components which can be accessed like an array.

4 An example extension grammar

In this section we give an example of how to specify the programming conventions for the object-oriented monitor library of Section 2.2 using attribute extension. The extension grammar for the library makes use of the base grammar interface given in Section 3.1. For each of the conventions listed in Section 2.2, we describe a number of specific checks and show how these can be specified in the extension grammar.

Figure 11 summarizes the notation used in extension grammars.

4.1 Monitor emulation

As noted in Section 2.2, any subclass to the library class Monitor is regarded as an emulated monitor. No checks are needed for this convention.

progdef A synthesized attribute whose defining equation may be overridden in the application program for individual syntax nodes.

error A synthesized string attribute which will be displayed as an error if its value is not equal to the empty string.

syn A synthesized attribute (defined in the node itself)

inh An inherited attribute (defined by the parent node)

eq An equation

eq all *NodeClass.attr* = *exp* Collective equation defining the value of the attribute *attr* of all children of type *NodeClass*.

The declaration of a synthesized attribute may include an equation. I.e.,

kind type attr = *exp;*

is equivalent to

kind type attr;
eq *attr* = *exp;*

Figure 11 *Notation used in extension grammar*

4.2 Entry procedure emulation

Entry methods are marked by /*= entry =*/

To control which methods should be regarded as entry methods, we require that each entry method is annotated by placing an attribute comment /*= entry =*/ before the method declaration. This is matched in the extension grammar by a program-defined attribute entry in the node class MethodDecl:

```
addto MethodDecl {
   progdef boolean entry = false;
}
```

Entry methods may only be declared in Monitor subclasses

Entry methods may only be declared in subclasses to Monitor. This is checked by adding an error attribute misplacedEntry to node class MethodDecl as follows:

```
addto MethodDecl {
   error string misplacedEntry =
      if entry and not inMonitorSubclass
      then “Misplaced entry method”
      else “”;
}
```

The definition of misplacedEntry uses (directly or indirectly) the attributes inMonitorSubclass and isMonitorSubclass. These attributes are defined as follows:

```
addto Declaration {
   inh boolean inMonitorSubclass;
};

addto DeclList {
   inh boolean inMonitorSubclass;
   eq all Declaration.inMonitorSubclass = inMonitorSubclass;
};

addto ClassDecl {
   syn boolean isMonitorSubclass =
      (superClassBinding() =/= null and
         (superClassBinding().globalName() = “MonitorLib:Monitor” or
         superClassBinding().isMonitorSubclass));

   eq DeclList.inMonitorSubclass = isMonitorSubclass;
};
```

The attribute isMonitorSubclass in ClassDecl is the central one, using the base grammar interface functions superClassBinding and globalName to find out if the ClassDecl represents a subclass to Monitor. Note that the extension grammar attribute isMonitorSubClass is accessed via the reference obtained from the function superClassBinding. This shows how the extension grammar can use references in the base grammar interface for propagating nonlocal information in the syntax tree, as was discussed in Section 3.2.

4.3 Entry procedure implementation

First/last statement of entry method must be enter/exit
An entry method should have a call to enter as its first statement and a call to exit as its last statement. To check this we use the same technique as was discussed in Section 3. Here is the full specification:

```
addto Method {
   error string missingEnter =
      if entry and not
         (MethodBody.globalMethodNameOfFirstStatement =
            “MonitorLib:Monitor:enter”)
      then “Missing call to enter”
      else “”:
   error string missingExit =
      if entry and not
         (MethodBody.globalMethodNameOfLastStatement =
            “MonitorLib:Monitor:exit”)
```

```
    then "Missing call to exit"
    else "":
}

addto MethodBody {
  syn string globalMethodNameOfFirstStatement =
    if cardinal()=0
    then ""
    else Statement[1].globalMethodName;
  syn string globalMethodNameOfLastStatement =
    if cardinal()=0
    then ""
    else Statement[cardinal()].globalMethodName;
}

addto Statement {
  syn globalMethodName = ""; Default equation
}

addto MethodCall {
  eq globalMethodName = methodDeclBinding().globalName();
  Overrides equation in superclass Statement
}
```

Calls to enter/exit must not occur at other places in the program.
To check that calls to enter and exit are not misplaced, we introduce two inherited attributes atFirstStatementOfEntry and atLastStatementOfEntry for Statements. These attributes will be true for the statements occurring first and last respectively in the body of an entry method. Statements can occur in many different contexts in a syntax tree, and all nodes which happen to have statement components need to define these attributes. To capture this behavior which is common to many different node classes, default defining equations are placed in the abstract superclass Node. MethodBody is the only node class overriding these equations. This is an example of the use of abstract superclasses in the node class hierarchy, as was discussed in Section 3.1.

```
addto Statement {
  inh boolean atFirstStatementOfEntry;
  inh boolean atLastStatementOfEntry;
}

addto Node {
  eq all Statement.atFirstStatementOfEntry = false; Default equation
  eq all Statement.atLastStatementOfEntry = false; Default equation
}

addto MethodBody {
  inh boolean inEntry
  eq all Statement.atFirstStatementOfEntry =
```

```
      (inEntry and Statement.childNo() = 1);
      Overrides equation in Node
   eq all Statement.atLastStatementOfEntry =
      (inEntry and Statement.childNo() = cardinal());
      Overrides equation in Node
}

addto MethodDecl {
   eq MethodBody.inEntry = entry;
}
```

Two error attributes are now introduced to check if enter and exit calls are misplaced:

```
addto MethodCall {
   error string misplacedEnter =
      if methodDeclBinding().globalName() = "MonitorLib:Monitor:enter"
         and not atFirstStatementOfEntry
      then "Misplaced call to enter"
      else "";
   error string misplacedExit =
      if methodDeclBinding().globalName() = "MonitorLib:Monitor:exit"
         and not atLastStatementOfEntry
      then "Misplaced call to exit"
      else "";
}
```

4.4 Data protection

Monitor must not contain any public data

To make sure that data in a monitor is manipulated only via the entry methods, we demand that monitors contain no public data. This is checked by adding an error attribute to VarDecl which uses the attribute inMonitorSubclass defined in Section 4.2:

```
addto VarDecl {
   error string publicData =
      if inMonitorSubclass and public()
      then "Data in monitor subclasses must not be declared public"
      else "";
}
```

Non-entry methods in monitors must not be public

This is to prevent clients from accessing data in a monitor without going via an entry method. This can be checked as follows:

```
addto MethodDecl {
   error string publicNonEntry =
      if not entry and public()
      then "Non-entry methods must not be declared public"
      else "";
}
```

Entry methods must be public
If the entry methods are not public, they are meaningless. This can be checked as follows:

```
addto MethodDecl {
   error string entryNotPublic =
      if entry and not public()
      then "Entry methods must be declared public"
      else "";
}
```

4.5 Conditional entry emulation

Calls to awaitChange should only occur in the appropriate places
Conditional entry of the monitor should be programmed by a statement

```
while condition do awaitChange
```

as the second statement in the entry method (directly after the call to enter). Calls to awaitChange must not occur in other places in the program. This can be checked in a similar way as checking for misplaced calls to enter and exit:

```
addto Statement {
   inh boolean inAwaitChangePosition;
   inh boolean inEntryBody;
}

addto Node {
   eq all Statement.inAwaitChangePosition = false; Default equation
   eq all Statement.inEntryBody = false; Default equation
}

addto MethodBody {
   eq all Statement.inEntryBody = inEntry;
      Overrides equation in superclass Node
}

addto WhileStmt {
   eq Statement.inAwaitChangePosition = (inEntryBody and childNo()=2);
      Overrides equation in superclass Node
}
```

```
addto MethodCall
   error string misplacedAwaitChange =
      if (methodDeclBinding().globalname() =
         "MonitorLib:Monitor:awaitChange") and
         not inAwaitChangePosition
      then "Misplaced call to awaitChange"
      else "";
}
```

4.6 Initialization

As noted in Section 2.2, initialization of internal data in the library class Monitor is done automatically at the creation of a Monitor instance. No checks are needed for this convention.

5 Implementation considerations

To implement an attribute extension system, one needs to address the implementation of the base grammar interface, the extension grammar evaluator, and the mapping of attribute comments to syntax nodes.

5.1 Implementation of the base grammar interface

To implement the functions in the base grammar interface efficiently, a straightforward solution is to represent the application program and the library interface as attributed abstract syntax trees, following the node class definitions in the base grammar interface. Information used to compute the base grammar functions can be stored as attributes (instance variables) in the tree. In particular, bindings between uses and declarations, subclasses and superclasses, etc., can be stored as reference attributes, making the implementation of superClassBinding and methodDeclBinding completely straight-forward.

We call the component computing the attribution a *base evaluator*. The base evaluator could be implemented by hand (for example by basing the implementation on an existing compiler), or it could be generated from a *base attribute grammar*. This has the additional benefit of providing an unambiguous declarative definition of the base grammar interface, serving as useful documentation when writing the extension grammars.

However, traditional attribute grammars [Knu68] cannot define reference attributes and are therefore not sufficiently expressive for these purposes. Instead, Door Attribute Grammars [Hed92, Hed94] can be used. Door AGs is a declarative extension to attribute grammars which explicitly supports reference attributes. This allows a name use site to be connected to its name declaration site, and conversely, a name declaration site to all its use sites. For object-oriented languages, subclasses can be connected to superclasses and vice versa.

The use of reference attributes to connect different parts of the syntax tree depending on the static-semantics, obviates the large "environment" attributes normally used in attribute grammars. Instead of foreseeing all information which possibly needs to be propagated from one site to another, and encapsulating it in a large environment attribute, the use of references allows attribute values to be propagated directly along the reference connections as need arises. This gives much simpler grammars, in particular for object-oriented languages where many bindings follow class hierarchies rather than the syntax tree hierarchy.

Due to the use of reference attributes, the Door AG evaluators cannot directly use standard attribute evaluation algorithms. In [Hed92] an algorithm for incremental evaluation is given where the major part of the evaluator is generated automatically from the grammar, but which relies on hand implementation of some critical parts in order to achieve an efficient incremental implementation. Algorithms for completely automatic evaluator generation, at least for exhaustive evaluation, are under development.

5.2 Implementation of the extension grammar evaluator

The goal of the extension grammar evaluator is to compute the values of the error attributes in the syntax tree for a program, as defined by the extension grammars for the libraries used by the program. For error attributes which have a non-empty string value, the evaluator outputs those values, similarly to how a normal compiler outputs compile-time error messages.

Demand evaluation

Demand evaluation is a very simple attribute evaluation technique which we expect to be the most practical for implementing extension grammar evaluators. It is a completely general technique handling any complexity of attribute dependencies, except for circular ones.

In demand evaluation, each attribute is implemented by a function and evaluation of the attribute is equivalent to calling the function. To evaluate the error attributes of a program, the evaluator simply scans the program and calls the function corresponding to each error attribute.

Demand evaluation is very simple to implement because it requires no dependency analysis of the attribute grammar. Because the grammar is defined as a class hierarchy, the implementation of demand attributes can be done by a straight-forward mapping from the grammar to virtual functions: a synthesized attribute corresponds exactly to a virtual function, and an inherited attribute to a virtual function with an extra argument [Hed89].

Another advantage of demand evaluation is that it requires no storage of attribute values - an attribute value is computed each time it is needed. This also has the benefit that in case attributes are not used, their values are never computed.

The drawback of demand evaluation is that it can be extremely time-inefficient: The call-tree for an attribute evaluation can be very large - it can in principle grow exponentially in the size of the program. However, in the context of library-specific checks, we expect very small call-trees since all the name analysis and type-checking information (accessed through the base grammar interface) is precomputed by the base evaluator. We therefore expect demand evaluation to be a practical technique in this context.

In case efficiency does become a problem, it is easy to extend the demand evaluation technique to cache critical attributes, in order to avoid repeated computations. If all attributes are cached, the evaluator will be time-optimal (no attribute is evaluated more than once), although with a larger constant overhead, and with the added space cost.

Data-driven evaluation

The alternative to demand evaluation is data-driven evaluation, where the attribute values are stored, and are evaluated in topological order, according to the attribute dependencies. In contrast to demand evaluation, *all* attributes are evaluated. There are many different algorithms for data-driven evaluation, handling different complexity of the attribute dependencies, e.g. OAG [Kas80]. Data-driven evaluators can be implemented without the overhead of function calls, and are therefore usually much faster in practice than demand evaluators.

Extension grammars differ from standard AGs in that equations may make use of non-local attributes, i.e., attributes accessed via the syntax node references returned by the functions in the base grammar interface. The data-driven algorithms for standard AGs can therefore not be applied directly to extension grammars. We do not expect it to be very difficult to adjust some popular algorithms to handle these non-local dependencies, but exactly how to do this remains to be investigated. (The non-local dependencies appearing in a Door AG are much more complex to handle since they are themselves a result of the attribute evaluation.)

To summarize, although demand-driven evaluation for general AGs is usually too inefficient, we expect it to be sufficiently efficient in the context of extension grammars because the complex computations are already taken care of by the base grammar interface. The alternative, data-driven evaluation, could also be considered, but in that case standard evaluation algorithms need to be adapted to handle non-local dependencies.

5.3 Mapping attribute comments to syntax nodes

The attribute comments used for annotating the application program need to be mapped to the corresponding syntax nodes. Earlier, we simply said that these comments are placed in front of the corresponding construct. However, it is necessary to be a bit more precise. Consider the following case:

```
/*= entry =*/ Element get() {...}
```

Here, the attribute comment `/*= entry =*/` is placed front of `Element` (the method return type), but we really mean it to apply to `get` (the method declaration). One solution to get an unambiguous mapping is to map an attribute comment for an attribute a to the root of the smallest subtree whose text representation immediately follows the comment and where that root declares a program-defined attribute a.

To map `/*= entry =*/` in the example above, we would first check the subtree "`Element`" of node class `ReturnType` and since it contains no program-defined attribute called `entry`, we would continue to check the subtree "`Element get() {...}`" of node class `MethodDecl`, which indeed contains a program-defined attribute called `entry`.

This mapping technique is simple to use and understand, but it is not completely general since there may be several subtrees (nested inside each other) whose textual representations start at the same point. If more than one of these subtrees has a program-defined attribute *a* in its root, only the attribute of the lowest of these roots can be accessed by the attribute comment. To be able to access the attributes in any of these roots, the mapping mechanism must be extended, for example as suggested below.

An upper and a lower root could either be of the same node class, for example in a nested left-recursive expression "`x + y + z`", or they could be of different node classes, for example in a statement list where the textual representations of the first statement and the complete list start at the same point.

In the case when the upper and lower roots are of the same node class, their textual representations must differ (otherwise, the grammar would be ambiguous). To handle this case, one can extend the mapping mechanism to allow attribute comments to be placed also *inside* a textual representation, e.g.:

```
x + y /*= a =*/ + z
```

The root is identified by looking at the roots of the subtrees whose text representation contains or immediately follows the point of the attribute comment, and selecting the lowest of these roots which has a program-defined attribute `a`.

This technique could also be used in many cases when the upper and lower roots are of different node classes. However, in some cases, such as in the statement list example, the textual representation of an upper root (the statement list) may consist of one or more segments, each corresponding to a lower root (a statement). In these cases, the upper root cannot be identified simply by placing the attribute comment inside the textual representation. A solution to this problem is to allow the attribute name to be qualified by a node class, e.g.:

```
/*= StatementList.a =*/
x = 3;
y = 4;
```

This attribute comment will reference the `a` attribute of the `StatementList` node, in contrast to a comment `/*= a =*/` or `/*= Statement.a =*/` which would have referenced the `a` attribute of the first `Statement`.

6 Related work

Other techniques aimed at supporting programming conventions include Minsky's Law-Governed Architecture and Meyer's CCEL system. Techniques like run-time assertions, preprocessing, and reflective programming are also interesting in this context.

6.1 Minsky's Law-Governed Architecture

Minsky's Law-Governed Architecture (LGA) [Min95] provides a framework for specifying and enforcing *regularities* of a software system. A regularity is a principle that must be observed everywhere in a system. Minsky's regularities include not only programming conventions, i.e. principles which have to be observed by the software source code, but also principles for the project database, the software process, and principles for how the regularities themselves (the "law") may be changed. The scope of the LGA system is thus wider than the programming conventions treated in this paper.

Interaction relations
To support the enforcement of programming conventions, the LGA system provides a number of predefined interaction relations, i.e. relations between different parts of the source code. For example, in an implementation of LGA for Eiffel [MiP94], there are predefined interaction relations like inherit(c1,c2) meaning that class c1 inherits directly from c2, and call(r, c1, f, c2) meaning that the routine r in class c1 contains a call (or access) to the routine or (variable) f of class c2. These interactions can be regulated by giving Prolog-like rules which state which interaction instances are legal.

The use of predefined interactions in LGA limits what kind of conventions can be expressed. In contrast, our technique based on an attribute grammar gives the user full freedom to specify any statically checkable conventions. For example, in the Eiffel implementation of LGA, all the interactions involve relations between classes and routines. There are no interactions involving the position of different statements. Thus, the LGA-Eiffel system cannot express the conventions 2, 3, and 4 in our monitor example of Section 4, e.g., that an entry procedure should start with a call to enter and end with a call to exit.

Properties and attributes
The LGA system allows the user to give properties to objects in the project database. For the LGA-Eiffel system, the database includes an object for each Eiffel class in the system. This ability to add properties can be compared to our technique of adding program-defined attributes. However, in LGA-Eiffel, the properties can only be added to the classes, whereas using our technique the attributes can be added to any linguistic construct. For example, in our monitor example, a program-defined attribute entry is added to the procedure construct. In LGA-Eiffel there is no possibility of adding properties to procedures. Since all of the conventions in Section 4 depend on this possibility, none of these conventions can be expressed in LGA-Eiffel.

Prohibitive versus permissive convention definition
The Prolog-like rules which regulate the interaction relations in LGA can be either *prohibition rules*, explicitly stating interaction cases which are disallowed, or *permission rules*, explicitly stating interaction cases which are allowed. Combinations of prohibition and permission rules are possible, giving prohibition rules precedence over the permission rules.

Our error attributes correspond to the prohibition rules, but we have no construct which corresponds directly to a permission rule. However, the set of permission rules for a given interaction can be reformulated as a single prohibition rule. While this gives the same effect, the possibility to define permission rules individually makes the rules more easy to read, and allows better modularization of rules. In our (so far limited) experience of defining conventions for libraries, we have found the prohibitive method of convention definition adequate for our needs. However, in [PaM96] a good example is given of permissive convention definition: stating various kinds of exceptions to the Law of Demeter by individual permission rules.

6.2 Meyer's CCEL language

Scott Meyers has developed a specification language CCEL (C++ Constraint Expression Language) for specifying programming conventions in C++ programs [MDR93, CM93b]. The goal of this research is very similar to ours, namely to be able to specify and enforce programming conventions, or *constraints*, as they are called in CCEL.

The CCEL language contains built-in attributed classes that correspond to the language constructs of C++. Objects of the CCEL classes thus correspond to constructs in a C++ program, such as individual classes and procedures. Subsets of CCEL objects can be identified by universal and existential quantifiers, and constraints can be expressed as assertions over the attributes of the selected objects.

In comparison to our approach, the CCEL built-in class hierarchy corresponds to a projection of a base grammar interface. Our approach is thus more general, since it allows constraints involving any language constructs to be expressed, not only those present in the CCEL language. For example, CCEL does not contain classes for statements. Thus, CCEL cannot express the conventions 2, 3, and 4 in our monitor example of Section 4, e.g., that an entry procedure should start with a call to enter and end with a call to exit.

Another difference is that CCEL contains no mechanism corresponding to our program-defined attributes. This is a severe drawback because it makes some conventions impossible to express. Actually, all of the conventions of our monitor example rely on the program-defined attribute entry, and none of these conventions can therefore be expressed in CCEL.

6.3 Run-time assertions

Assertions were originally intended for program verification and rigorous program construction. However, run-time-checked assertions can also be used within a library to check that certain programming conventions are followed by the application program using the library. This technique, often referred to as *design by contract*, is used in Eiffel [Mey92] which supports pre- and postconditions of methods, and class invariants. In particular, method preconditions can be used to check that an application program calls a library method with appropriate parameters and in an appropriate program state. However, many programming conventions cannot be captured by run-time assertions in the library. For example, none of the conventions in Section 4 can be checked by assertions in the library code. Thus, run-time assertions and library-specific static checks complement each other as techniques for supporting programming conventions.

6.4 Preprocessing

It is common to extend a language by defining macros which are expanded to base language code by a preprocessor, or to define a complete domain-specific language with a preprocessor which translates application programs to a base language. Although this may be convenient for some problems, there are also several disadvantages. Preprocessing in general has the disadvantage that errors which occur during compile-time or run-time will be related to the expanded code which the application programmer does not recognize. For macro expansion there are additional problems. For example, there is no checking that the macros are used in the intended way by the application program. Since macros usually work at the lexical level, unintended use can result in very strange errors.

6.5 Reflective programming

In reflective programming, a program can access and manipulate a representation of itself [Smi84]. The use of reflection in object-oriented languages has recently received considerable attention, in particular for CLOS for which a Meta Object Protocol (MOP) allows a program to change the method invocation mechanism [KRB91]. Also for other languages, reflective facilities have been designed, for example mechanisms for redefining method lookup in Smalltalk [FJ89] and C++ [CM93a]. In relation to the monitor example discussed in Section 2, reflection could be used to redefine method lookup for monitor objects in order to encapsulate incoming messages by calls to enter and exit.

Similar to the use of submethoding discussed in Section 2.3, reflective facilities add power to a language, making certain behavior easier to program than in a traditional non-reflective language and it may therefore reduce the need for programming conventions.

7 Conclusions

In constructing programming libraries, it is common that application programs must obey certain programming conventions in order for the library to work correctly. As was illustrated in Section 2, the base language has a dramatic influence on the complexity of the needed programming conventions. An object-oriented language gives much better support than a procedural language, and an advanced concept like submethoding in BETA gives even more support. However, even with a very powerful base language, there are some conventions that cannot be captured directly in the base language.

To extend the possibilities of enforcing programming conventions, we have suggested a technique of adding library-specific static checks using an attribute grammar notation. The library-specific checks can make use of an existing base grammar interface, capturing name bindings and types. The use of references in the base grammar interface which explicitly connect named entities makes addition of library-specific attributes simple, avoiding the problems of large attributes in standard attribute grammars.

We have shown how the technique can be applied to enforce the conventions of a monitor library for an object-oriented base language. Our grammar-based architecture with a base language interface and an extension grammar with program-defined attributes gives a generality to our approach which is not present in other approaches with a similar goal of specifying and enforcing programming conventions, like Minsky's law-governed architecture for Eiffel, and Meyers' CCEL language for C++. Neither of these can specify the conventions in our monitor example.

Although our technique is based on attribute grammars, which might be considered to be outside the skills of an ordinary programmer, we think the principles are not very difficult to grasp for people skilled in object-oriented programming. We expect that in any team producing such complex software as object-oriented libraries and frameworks, there will be people who can easily master the technique of writing extension grammars. The users of the frameworks do not need to be as skilled; they do not need to understand the attribute extension mechanism. They could rely on documentation of the programming conventions written in natural language, similar to how they use a natural language description of the base language, rather than any formal description of it.

An interesting possibility for the use of attribute extension would be to apply it to the documentation of design patterns [GHJV94]. In current practice, the application of design patterns is at best documented informally in the code. This lack of traceability leads to problems in understanding how the patterns are applied and how to maintain the code without breaking existing patterns [Sou95, Bos96]. The patterns are thus a kind of programming conventions which are informally applied. By using attribute comments one could annotate program code to formally document how a design pattern is applied. Some patterns include conventions for how methods should be called. For example, in the Observer pattern for change notification [GHJV94], an object playing the role of a "Subject" should call a special notification method every time its state has changed in order to notify all its dependents of the change. Such a convention could be described formally in an extension grammar.

Acknowledgements

I am grateful to Bo Sandén, Kai Koskimies, and the anonymous reviewers for providing valuable comments on earlier drafts of this paper.

References

[Bos96] J. Bosch. Design Patterns as Language Constructs. To appear in *Journal of Object-Oriented Programming*.

[Bri75] P. Brinch Hansen. The Programming Language Concurrent Pascal. IEEE Transactions on Software Engineering SE-2 (1). June 1975. pp 199-207.

[CM93a] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'93)*, Kaiserslautern, LNCS 707, pp 482-501, July 1983.

[CM93b] A. Chowdhury and S. Meyers. Facilitating Software Maintenance by Automated Detection of Constraint Violations. In *Proceedings of the Conference on Software Maintenance*, 1993.

[Deu89] P. Deutsch. Design Reuse and Frameworks in the Smalltalk-80 System. In T. J. Biggerstaff and A. J. Perlis (editors), Software Reusability. Vol II. Applications and Experience. ACM Press 1989.

[DMN68] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA 67 common base language*. NCC Publ. S-2, Norwegian Computing Centre, Oslo, May 1968.

[FJ89] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. *Proceedings of OOSPLA'89*, New Orleans, LA, pp 327-335, October 1989.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Hed89] G. Hedin An object-oriented notation for attribute grammars. In S. Cook, editor, *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, BCS workshop Series, pp 329-345, Nottingham, U.K., July 1989. Cambridge University Press.

[Hed92] G. Hedin. *Incremental semantic analysis*. PhD thesis, Lund University, Lund, Sweden, 1992.

[Hed94] G. Hedin. An overview of door attribute grammars. In *Proceedings of the 5th international conference on Compiler Construction (CC'94)*, LNCS 786, pp 31-51, Edinburgh, April 1994. Springer-Verlag.

[Hoa74] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *CACM* 10 (17). Oct 1974. pp 549-557.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 4(2):22-35 June/July 1988.

[Kas80] U. Kastens, Ordered Attributed Grammars. *Acta Informatica*, 13:229-256, 1980.

[Knu68] D. E. Knuth. Semantic of context-free languages. Matchematical Systems Theory, 2(2):127-145, June 1968.

[KRB91] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*, MIT Press 1991.

[LH89] K. J. Lieberherr and I. M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, Sept 1989, pp 38-48.

[LM81] M. Löfgren and B. Magnusson. An extension of Simula for concurrent execution. In *Proceedings of the 9th Simula Users' Conference*. 1981, Geneva.

[Mag96] B. Magnusson. A kernel with preemption for real time programming in Simula. Lund Institute of Technology, Sweden. June 1996.

[MDR93] S. Meyers, C. K. Duby, and S. P. Reiss. Constraining the Structure and Style of Object-Oriented Programs. *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, April 1993.

[Mey92] B. Meyer. Applying "Design by Contract". *IEEE Computer*. pp 40-51. October 1992.

[Min95] N. H. Minsky. Law-Governed Regularities in Object Systems; Part 1: an Abstract Model. TAPOS 4(1) 1995.

[MiP94] N. H. Minsky and P. Pal. Law-Governed Regularities in Object Systems; Part 2: the Eiffel Case. Technical Report LWWW.CSR-TR-22, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1994.

[MMN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. ACM Press, 1993.

[PaM96] P. Pal and N. H. Minsky. Imposing the Law of Demeter and Its Variations. Proceedings of the TOOLS Conference, Aug 1996, Santa Barbara, CA, USA.

[Smi84] B. C. Smith. Reflection and Semantics in Lisp. *Proceedings of the 1984 ACM Principles of Programming Languages Conference*, pp 23-35. 1984.

[Sou95] J. Soukup. Implementing Patterns. In *Pattern Languages of Program Design*, Coplien, Schmidt (eds), pp 395-412. Addison-Wesley. 1995.

[Vau75] J. Vaucher. Prefixed procedures: A structuring concept for operations, *INFOR*, Vol. 13, No. 3, October 1975.