Enforcing programming conventions by attribute extension in an open compiler

Görel Hedin

LU-CS-TR:96-171
LUTEDX/(TECS-3068)/1-18/(1996)

Also published in: Proceedings of NWPER'96, Nordic Workshop on
Programming Environment Research, L. Bendix, K. Nørmark,
and K. Østerbye (eds), Aalborg, Denmark, May 1996.
Aalborg University, Technical Report R-96-2019.

Department of Computer Science

Lund Institute of Technology
Lund University

Box 118, S-221 00 Lund, Sweden

Enforcing programming conventions by attribute extension in an open compiler

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 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 suggests a technique for this by means of an extensible attribute-grammar based compiler.

Keywords: application languages, extensible languages, reusability, frameworks, object-oriented programming, attribute grammars, open 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 an application-oriented language, because abstraction mechanisms such as classes and procedures allows the modelling of application-specific concepts. In object-oriented frameworks this idea is 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 [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 extend an open compiler 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 application-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 will work best with a statically typed object-oriented base language such as Simula, BETA, Eiffel, C++, or Java. With a statically typed language we mean a language where named entities (variables, classes, methods, etc.) have compile-time types[1]. This gives a basic semantic framework necessary for being able to add library-specific checks. Our approach would also be useful for procedural languages, but it is more attractive to use an object-oriented language which requires less programming conventions in the first place.

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.

Attribute extension

The implementation technique we propose is to build an open compiler which makes the static-semantics of a base language program available as an attributed syntax tree, specified in an object-oriented form supporting reference and collection-valued attributes, by means of Door Attribute Grammars (a space-efficient declarative representation, initially intended for efficient incremental

1. The compile-time types allow most type checking to be done at compile time. However, the compile-time types are often less precise than the run-time types, and some type-checking may therefore need to be done at run time.

attribute evaluation) [Hed92, Hed94]. The open compiler supports *attribute extension* by allowing additional attributes and equations to be added to the language specification, and by allowing programs in the base language to be annotated with user-specified attribute values.

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 and the architecture of an open compiler enabling such extensions. In section 4 we show the specification of a simple extension using this technique. Section 5 relates our approach to other techniques for language extension. Section 6 summarizes the results and discusses possibilities for further research.

2 The monitor example

A *monitor* is a class or abstract data type where special *entry procedures* guarantee mutually exclusive access to data in the class/ADT instance. In this section we will look at how a monitor construct can be simulated by a library, what programming conventions need to be followed for the monitors to work correctly, and how the base language influences the level of support.

2.1 A procedural approach

The basic functionality needed for a monitor can be realized by a procedural library, given in figure 1. (This interface is a simplified version of a teaching package for concurrent programming used at Lund Institute of Technology.)

```
type record MonitorVariable ... (* a kind of semaphore *)

procedure InitMonitor (M: ref MonitorVariable) ...
    (* Initializes the monitor variable *)
procedure EnterMonitor (M: ref MonitorVariable) ...
    (* Waits until the monitor is free, then enters it (locks it) *)
procedure ExitMonitor (M: ref MonitorVariable) ...
    (* Leaves the monitor (unlocks it). Also causes processes awaiting change to be put on
queue for entering the monitor. *)
procedure AwaitMonitorChange (M: ref MonitorVariable) ...
    (* Leaves the monitor and waits until some other process has visited the monitor. Then
enters the monitor again. *)
```

Figure 1 *Procedural library for basic monitor functionality*

When using this library, the following conventions should be followed:

- A monitor variable instance should be placed in the record holding the data protected by the monitor.
- Access to the protected data should be done only via *entry procedures*. These procedures should be realized by ordinary procedures which have a

call to EnterMonitor as their first statement and ExitMonitor as their last statement.

- Conditional entry of the monitor should be programmed by a statement
  while *condition* do AwaitMonitorChange(M);
  as the second statement in the entry procedure.
- A call to InitMonitor must be made before any call to an entry procedure for the monitor.

Figure 2 outlines an example application program with a FIFO bounded buffer monitor which follows the above conventions.

```
(* monitor *) record type FIFOMonitor
    M: ref MonitorVariable;
    BB: ref BoundedBuffer;
end record type;

(* init *) procedure InitFIFO (B: ref FIFOMonitor);
    B.M :- new MonitorVariable;
    InitMonitor(B.M);
    B.BB :- new BoundedBuffer;
    InitBuffer(B.BB);
end procedure;

(* entry *) procedure put (B: ref FIFOMonitor, E: ref Element);
    EnterMonitor (B.M);
    (* condition *) while BoundedBufferFull(B.BB) do AwaitMonitorChange(B.M);
    BoundedBufferAddAsLast(B.BB, E);
    ExitMonitor (B.M);
end procedure;

(* entry *) ref Element procedure get (B: ref FIFOMonitor);
    EnterMonitor (B.M);
    (* condition *) while BoundedBufferEmpty(B.BB) do
                        AwaitMonitorChange(B.M);
    return BoundedBufferRemoveFirst(B.BB);
    ExitMonitor (B.M);
end procedure;
```

Figure 2 *Procedural application program containing a FIFO monitor*

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:

- Omission of a call to EnterMonitor may lead to unsynchronized updates of the buffer, leading to inconsistencies and to run-time errors at completely different places in the program.
- Access to monitor data outside of an entry procedure gives similar errors.
- Omission of a call to ExitMonitor may lead to deadlock.
- Calls to AwaitMonitorChange outside an entry procedure will lead to a run-time error.
- Omission of the call to InitMonitor will lead to a run-time error in the first call to EnterMonitor.

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. For example, by placing the monitor variable somewhere outside the monitor data record, or by placing the EnterMonitor and ExitMonitor calls at the entry procedure call sites rather than inside the entry procedures.

2.2 An object-oriented approach

By using an object-oriented language, 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 3 and 4. In particular, the existence of the monitor variable and the monitor initialization is completely hidden from the application programmer.

```
class Monitor
   method Enter...
   method Exit...
   method AwaitChange...
end class;
```

Figure 3 *Library for abstract monitor class*

```
class FIFOMonitor extends Monitor
   BB: ref BoundedBuffer :- new BoundedBuffer.init;

   (* entry *) method put (E: ref Element)
      Enter;
      (* condition *) while BB.Full do AwaitChange;
      BB.AddAsLast(E);
      Exit;
   end method;

   (* entry *) ref Element method get
      Enter;
      (* condition *) while BB.Empty do AwaitChange;
      return BB.RemoveFirst;
      Exit;
   end method;
end class;
```

Figure 4 *Object-oriented application program containing a FIFO monitor*

In the object-oriented solution, fewer programming conventions are needed than in the procedural case. 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.

The realization of the monitor construct as a class gives a syntactic relation between the entry methods, the monitor variable, and the protected data. This makes the remaining programming conventions easier to describe and specify than in the procedural case.

2.3 A refined approach using submethoding

We can do even better by using an object-oriented language which supports submethoding as well as subclassing, as present in BETA [MMN93]. A *submethod* extends another method in the following way: The supermethod may contain a statement inner which causes the code of the submethod to be executed. 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 principle examples used. The use of subclassing and submethoding for modeling monitors is also treated in [LM81] and [MMN93].

Figure 5 and 6 illustrate how submethods can be used in the class library to encapsulate the calls to Enter and Exit in an abstract method Entry. In the example, we also make use of unlimited block structure (as present in Simula and BETA) and declare the condition as a virtual method Condition within the abstract method Entry. This allows us to encapsulate also the while loop with the call to AwaitChange within the method Entry. In the application program, the Condition method is overridden with the appropriate actual condition in the submethods.

```
class Monitor
    method Entry
        boolean virtual method Condition
            return false;
        end method;

        Enter;
        while Condition do AwaitChange;
        inner;
        Exit;
    end method;
end class;
```

Figure 5 *Class library with abstract method*

```
class FIFOMonitor extends Monitor
    BB: ref BoundedBuffer :- new BoundedBuffer.init;

    method put extends Entry (E: ref Element);
        method Condition return BB.Full end;
        BB.AddAsLast(E);
    end method;

    ref Element method get extends Entry
        method Condition return BB.Empty end;
        return BB.RemoveFirst;
    end method;
end class;
```

Figure 6 *Object-oriented application program using submethoding*

By using this powerful base language we have actually managed to encapsulate most of the programming conventions in the library. However, there are still a few conventions which could be useful to express using library-specific semantic checks. For example:

- In subclasses to class Monitor, submethods of Entry are the only features which may be accessed from other objects.

By enforcing the above convention one can avoid that the application programmer unintentionally forgets to specify an entry method as a submethod of Entry.

2.4 Summary

We have seen in this example how the choice of base language dramatically influences the number of programming conventions which need to be followed by an application programmer. However, even a very advanced language like BETA can not encapsulate 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

3.1 Door Attribute Grammars

To do library-specific static-semantic 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 tool which performs name- and type analysis for the base language, and to which library-specific checks can be easily added. When running the tool on a given application program, the checks for each of the used libraries should be performed by the tool.

Our approach is to build the extensible tool using Door AG technology. A Door AG [Hed92, Hed94] is an extension of standard attribute grammars which allows references and collections of references to be specified as part of a syntax tree attribution. This allows a name use site to be connected to its name declaration site, and conversely, a name declaration site to be connected 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. New attributes can easily be added and propagated along the reference attribute connections without needing to change existing attributes. For example, a new attribute can be added to the class construct, and propagated to all subclasses by means of a single equation. (An equation defines the value of an attribute in terms of other attributes.) Doing the equivalent in a standard attribute grammar would require changing or adding many large environment attributes.

Another advantage of Door AGs is that they are based on object-oriented attribute grammars where nonterminals and productions are modelled as *node*

classes [Hed89]. A standard AG with nonterminals and productions corresponds to an OO AG with a two-level class hierarchy: each nonterminal is a superclass of its production subclasses. By using the full power of OO AGs, class hierarchies of any level can be constructed, and 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.

3.2 Specification of library grammar units

With *attribute extension* we mean the technique for extending a base language Door AG with additional attribute declarations and equations. Figure 7 depicts how a base language Door AG is extended with library-specific grammar units.

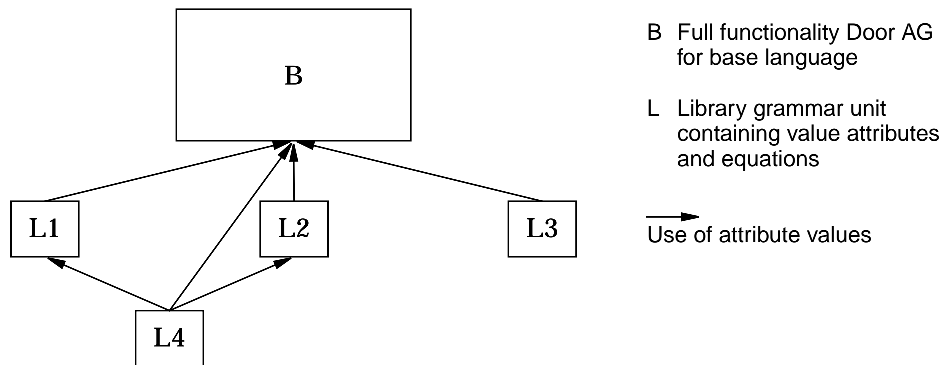

Figure 7 *Extension of Door AG by library grammar units*

A library grammar unit adds attribute declarations and equations defining those attributes to the node classes of the base language. The equations may make use of attribute values in the base language grammar, and also in other library grammar units.

Currently, we are considering only allowing standard AG attributes (value attributes) to be defined in the library units, i.e. disallowing the definition of new reference attributes or collections. However, reference attributes and collections in the base language grammar may be accessed freely by the equations in the library unit. For most library units, we think it is sufficient to be able to define value attributes, since the complex name analysis is done already in the base language grammar. This restriction allows us to use a simple evaluation technique for the library units, as will be discussed in section 3.3.

Program-defined attributes

To allow a given application program to control the values of individual attribute instances, the notion of *program-defined attributes* is introduced. A program-defined attribute is a local attribute which has an equation defining its

value, just like other attributes. In addition, it is possible to override this definition by annotating the program with program-defined equations. A program-defined equation is attached to a specific language construct instance in the program and will override the corresponding equation given in the library grammar unit for that particular language construct instance.

For specifying program-defined equations we will use structured comments with the following syntax:

```
(** 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 (1, 2, 3, ...), or a literal text value.

As an example, consider the object-oriented monitor library in section 2.2. Here we wish to model the concept of an entry method by attaching a boolean attribute entry to the production Method in the base language:

```
addto Method {
   progdef entry: boolean;
   equation entry = false;
};
```

This specification says that the default value of entry is false for each method. In the application program using the monitor library, the entry attribute can be given the value true for individual methods in the application program as follows:

```
(** entry = true **) method put (E: ref Element) ...
(** entry = true **) ref Element method get ...
```

Because it is very common for program-defined attributes to be boolean, we will allow the program-defined equations for such attributes to be abbreviated to simply the name of the attribute, meaning that the attribute is defined to true. Thus, the following specification is equivalent to the one above:

```
(** entry **) method put (E: ref Element) ...
(** entry **) ref Element method get ...
```

Error attributes

The “output” of a library grammar unit is a set of error messages attached to language constructs in the application program using the library. To support this we introduce the notion of an *error attribute*. An error attribute is a string-valued attribute which should be given an appropriate error text as value if there is a library-specific error, and the empty string-value 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 Method as follows:

```
addto Method {
   error missingEnter: string;
   equation missingEnter =
         if entry and not (first statement is a call to Enter)
         then “Missing call to Enter”
         else “”:
};
```

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 error attribute missingEnter 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.

One way of supporting this is to define an attribute globalname for each named entity construct. For example, the global name for the Enter method could be MonitorLib:Monitor:Enter, where MonitorLib is the name of the library containing the Monitor class. Since most library-specific checks will need to refer to named entities, it is most convenient if the global name attributes are defined in the base language grammar.

Using global names, the equation defining the error attribute missingEnter above could be defined more precisely as follows:

```
   equation missingEnter =
         if entry and not
            (Body.globalnameOfFirstStatement = “MonitorLib:Monitor:Enter”)
         then “Missing call to Enter”
         else “”:
```

where “Body” is the body component of the method construct, and globalnameOfFirstStatement is a synthesized attribute of Body which is the global name of the method called in the first statement of the body, or the empty string if the first statement is something else than a method call.

3.3 Design of the attribute extension tool

We are currently only in the initial stage of an actual implementation of an attribute extension tool, and will here only outline our design of it. We intend to implement the attribute extension tool as a part of our generic incrementally compiling environment, Orm [MHM+90].

The Orm system
The Orm system maintains a complete attribution of the program, with name bindings, type information, and error messages. Static-semantic checking is performed incrementally, by updating the attribution after each single edit step (such as insert/remove declaration/statement). Incremental feedback on static-

semantic errors are given after each such edit step by highlighting the erroneous constructs.

In implementing attribute extension, it would of course be desirable to also have incremental updating of library-specific errors, in the same way as for base language errors. However, implementation of the incremental updating can currently not be done completely automatically for Door AGs, so initially, we will use a variant of exhaustive evaluation for the library-specific checks. This way, the implementation of the evaluator for the library grammar units can be done completely automatically.

The base language grammar will be evaluated incrementally, just as in the existing Orm system. In addition, the user will be able to invoke library-specific checking by a menu command, at which point the exhaustive library-specific evaluator will be run.

The library-specific evaluator

As mentioned earlier, the library grammar units contain only standard AG elements, i.e. value attributes and their defining equations. Although the library grammar units may access Door AG specific attributes in the base language grammar, e.g. reference attributes and collections, we can still use standard AG evaluation techniques for the library grammar units. This is because the library-specific evaluation is done exhaustively for a given base language attribution, which can thus be regarded as constant during the library-specific evaluation.

We will use a very simple technique for the library-specific evaluation, namely an exhaustive scan over the program, evaluating each error attribute by means of *demand evaluation*. In demand evaluation, an attribute is implemented by a function and evaluation of the attribute is equivalent to calling the function. Demand evaluation is very simple to implement because it requires no dependency analysis of the attribute grammar. Because Door AGs are based on OO AGs, the implementation of demand attributes can be done by a straightforward mapping from the grammar to virtual functions: a synthesized demand 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. 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, for two reasons: 1) Any access to a base grammar attribute will truncate the call-tree since the values of these attributes are stored in the syntax tree. 2) The library-specific equations will be quite simple since all the complex functionality of name analysis and type checking is already present in the base grammar.

Thus, although we intend to use very simple evaluation techniques, we expect the performance to be acceptable for practical use.

4 An example specification

As an example of library-specific checks, we will look at how the programming conventions for the object-oriented monitor library of section 2.2 can be specified. The complete specification of the library grammar unit is given in the appendix.

Base language grammar

The base language grammar contains two general node classes: Node - the abstraction of any syntax node in a base language program, and its subclass Descendant - the abstraction of any Node except for the root node. The rest of the classes are subclasses of Descendant and correspond to nonterminals or productions.

The node class Declaration models a general declaration in the base language, and has subclasses like Class and Method. Each Declaration has an attribute globalname, holding the global name of the declaration as discussed in section 3.2, and a boolean attribute protected (see convention 6 below). Node classes which access named entities, like MethodCall and SuperClass, have a reference attribute DeclBinding to the corresponding Declaration node.

The following notation for node classes is used:

NodeClass [**:** *SuperClass*] **::=** [**(** *SonNodes* **)**] **{** *AttributeDecls* **}**

The specification below shows the parts of the base language grammar that are used by the library grammar unit.

```
Node ::= { }
Descendant: Node ::= { }

Declaration: Descendant::= {
      globalname: string;
      protected: boolean; }
Method: Declaration ::= (MethodId FormalParamList OptReturnType Body) { }
Body: Descendant ::= (list of Statement) { }
Statement: Descendant::= { }
MethodCall:Statement ::= (MethodId ActualParamList) {
      DeclBinding: ref Declaration; }
WhileStmt: Statement ::= (cond:Expression doPart:Statement) { }
Class: Declaration ::= (OptSuperClass ClassId MethodList) { }
OptSuperClass: Descendant ::= { }
SuperClass: OptSuperClass ::= (SuperId) {
      DeclBinding: ref Declaration; }
MethodList: Descendant: list of (Method) { }
```

Convention 1: Entry methods are marked by (entry **)**

To follow this convention, the application program should put a structured comment (** entry **) before each method declaration intended to be an entry method. This is matched in the library grammar unit by a program defined attribute entry in the node class Method.

Convention 2: First/last statement of entry method must be Enter/Exit
To follow this convention, an entry method of an application program must have a call to Enter as its first statement and a call to Exit as its last statement. This convention is checked by two error attributes in node class Method: missingEnter and missingExit.

Convention 3: Enter and Exit must not occur in other positions
I.e., the methods Enter and Exit must not be called in other positions than as stated by convention 2. To check this, two error attributes misplacedEnter and misplacedExit are added to node class MethodCall. A boolean attribute inBodyOfEntry, which is true if a statement is in the body of an entry method, is propagated down from a Method to its body statements. For all other statements, the attribute inBodyOfEntry is defined as false. This default specification is given in the general node classes Node and Descendant.

Convention 4: AwaitChange must only appear in condition position
I.e., a call to AwaitChange must only occur as the do-part of a while-statement which occurs as the second statement of an entry procedure. This is checked by an error attribute misplacedAwaitChange in the node class MethodCall. A boolean attribute inAwaitChangePosition, which is true if a statement is in the correct position for a call to AwaitChange, is propagated down in a similar way as the inBodyOfEntry attribute of convention 3.

Convention 5: Entry methods must be declared in Monitor subclass
I.e., the application program must only mark a method by the structured comment (** entry **) if it is a method of a subclass to Monitor. This is checked by an error attribute misplacedEntry in node class Method. To define this attribute, synthesized attributes isMonitor and isMonitorSubclass in node class Class are accessed by subclasses via the DeclBinding reference attribute connecting a subclass with its superclass. An inherited attribute inMonitorSubclass, which is true if a method is declared in a subclass to Monitor, is propagated down from a class to its declared methods in a similar way as the inBodyOfEntry attribute of convention 3.

Convention 6: Non-entry methods in Monitor subclasses must be protected
I.e., if the application program declares methods other than entry methods in Monitor subclasses, these methods must be declared as protected, i.e., not accessible from outside instances of the Monitor subclass. (We assume that the base language has such a *protected* mechanism, similar to the ones of Simula and C++.) This will hinder access of data in a Monitor instance other than via entry procedures. To check this convention, an error attribute nonProtectedNonEntry is added to node class Method. This attribute is defined using the attribute protected present in the base language grammar, and the attribute inMonitorSubclass defined in convention 5.

5 Related work

Related techniques for supporting programming conventions include run-time assertions and language extension techniques like preprocessing and reflective programming. However, we are not aware of any other approaches to enforcing programming conventions by adding library-specific static-semantic checks.

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 *programming by contract*, is used in Eiffel [Mey88] 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. On the other hand, there are also conventions that cannot be checked by static checks, and where run-time assertions are needed. Thus, run-time assertions and library-specific static checks complement each other as techniques for supporting programming conventions.

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 application-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.

Reflective programming

In reflective programming, a program can access and manipulate a representation of its own state [Smi84]. The use of reflection in object-oriented languages has recently received considerable attention, in particular for interpreted languages. Also for compiled languages, reflective facilities have been designed, for example mechanisms for redefining method lookup in Smalltalk [FJ89] and C++ [CM93]. 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 conventional non-reflective language and it may therefore simplify the needed for programming conventions.

6 Conclusions and future work

In constructing programming libraries, it is usual 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 needed number of 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 to enforce 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 attribution, capturing name bindings and types. The use of reference attributes in the base grammar explicitly connects named entities and 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.

This work is still in an initial stage, and future work includes full implementation of a tool based on the presented ideas, and more case studies on existing libraries.

References

[CM93] 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.

[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.

[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.

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

[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.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[MHM+90] B. Magnusson, G. Hedin, and S. Minör, et al. An overview of the Mjølner/Orm environment. In J. Bezivin et al., editors, *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, pp 635-646, Paris, June 1990. Angkor.

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

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

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

Appendix Specification of Monitor library-specific checks

Convention 1

```
addto Method {
    progdef entry: boolean = false; }
```

Convention 2

```
addto Method {
    error missingEnter: string =
        if entry and not
            (Body.globalnameOfFirstStatement =
                "MonitorLib:Monitor:Enter")
        then "Missing call to Enter"
        else "":
    error missingExit: string =
        if entry and not
            (Body.globalnameOfLastStatement =
                "MonitorLib:Monitor:Exit")
        then "Missing call to Exit"
        else "": }

addto Body {
    syn globalnameOfFirstStatement: string =
        if Statement.cardinal=0
        then ""
        else Statement[1].globalname;
    syn globalnameOfLastStatement: string =
        if Statement.cardinal=0
        then ""
        else Statement[cardinal].globalname; }

addto Statement {
    syn globalname := ""; (* Default *) }

addto MethodCall {
    eq globalname = DeclBinding.globalname; }
```

Convention 3

```
addto Descendant {
    inh inBodyOfEntry }

addto Node {
    eq all Descendant.inBodyOfEntry = false; }

addto Method {
    eq Body.inBodyOfEntry = entry; }

addto Body {
    eq all Statement.inBodyOfEntry =
        inBodyOfEntry }

addto MethodCall {
    error misplacedEnter: text =
        if (DeclBinding.globalname =
                "MonitorLib:Monitor:Enter")
            and not inBodyOfEnter
        then "Misplaced call to Enter"
        else "";
    error misplacedExit: text =
        if (DeclBinding.globalname =
                "MonitorLib:Monitor:Exit")
            and not inBodyOfEnter
        then "Misplaced call to Exit"
        else ""; }
```

Convention 4

```
addto Descendant {
    inh inAwaitChangePosition: boolean;

addto Node
    eq all Descendant.inAwaitChangePosition =
        false;

addto WhileStmt
    eq DoPart.inAwaitChangePosition =
            (inBodyOfEnter and sonNo=2);

addto MethodCall
    error misplacedAwaitChange: text =
        if DeclBinding.globalname =
            "MonitorLib:Monitor:AwaitChange"
        then
            if not inAwaitChangePosition then
                "Misplaced call to AwaitChange"
            else ""
        else ""; }
```

Convention 5

```
addto Descendant {
    syn isMonitor: boolean = false; (* Default *)
    syn isMonitorSubclass: boolean = false;
            (* Default *) }

addto Class {
    eq isMonitor =
            (globalname = "MonitorLib:Monitor");
    eq isMonitorSubclass =
        OptSuperClass.isMonitor or
        OptSuperClass.isMonitorSubclass; }
```

```
addto SuperClass {
    eq isMonitor = DeclBinding.isMonitor;
    eq isMonitorSubclass =
        DeclBinding.isMonitorSubclass; }

addto Descendant {
    inh inMonitorSubclass: boolean; }

addto Node {
    eq all Descendant.inMonitorSubclass =
        false; }

addto Class {
    eq MethodList.inMonitorSubclass =
        isMonitorSubclass; }

addto MethodList {
    eq all Method.inMonitorSubclass =
        inMonitorSubclass; }

addto Method {
    error misplacedEntry: text =
        if entry
        then
            if not inMonitorSubclass
            then "Misplaced entry procedure"
            else ""
        else ""; }
```

Convention 6

```
addto Method {
    error nonProtectedNonEntry: text =
        if InMonitorSubclass
        then
            if not entry
            then
                if not protected
                then "Missing declaration of this
                        method as protected"
                else ""
            else ""
        else ""; }
```

Notation

progdef A local attribute whose defining equation may be overridden in the application program for individual node instances

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

syn A synthesized attribute (may be accessed by the parent node)

inh An inherited attribute (must be defined by the parent node)

eq An equation

eq all NodeClass.Attr = exp Collective equation defining the value of the attribute Attr for all son nodes of class NodeClass

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

kind attr: type = exp;

is equivalent to

kind attr: type;
eq attr = exp;