ELSEVIER

Information Processing Letters 48 (1993) 163–170

Information Processing Letters

A description of dynamic behavior for compilers based on object oriented modeling

DongGill Lee [*,a,b], Kwang-Moo Choe [b], Taisook Han [b]

[a] *Software Development Environments Section, Electronics and Telecommunications Research Institute, P.O. Box 8, Dae Dog Dan Ji, Daejon, South Korea*

[b] *Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-dong, Yusong-gu, Daejon 305-701, South Korea*

(Communicated by K. Ikeda)
(Received 11 September 1992)
(Revised 13 September 1993)

Key words: Compilers; Language semantics; Object-oriented modeling; Dynamic behavior; Attribute state; Object state; Finite state machine

1. Introduction

The *syntax-directed translation* [1,7] technique based on attribute grammar has been the most commonly used method for compiler development. However, as a design of compiler with syntax-directed translation depends on the parsing methods, the conceptual structure of the compiler does not correspond well to the actual modules which implement its conceptual structure. As a result, the developed compiler may have the structure with less modularity. Moreover, as the semantics in modern high-level programming languages becomes more complicated, compilers are also getting complicated. These aspects may lead to difficulties in compiler development and maintenance for such languages.

To enhance modularity and reusability of software, various methodologies have been developed and used to support *object-oriented modeling* as a design technique in the software engineering. With this tendency, it is necessary to introduce the object-oriented modeling technique to compiler development to improve the modularity of compilers. However, there has been little attention on object-oriented modeling for compiler design.

For the object-oriented modeling for compilers, we use the concepts of the *object model* [6] which was originally suggested by Rumbaugh. The object model does not describe compilers completely because it represents only the *static structure* of a compiler. In this paper, we propose a method to describe *dynamic behavior* of the object model for the compilers.

2. Considerations for object-oriented modeling

We use the same terminology for the object model that is defined by Rumbaugh [6]. A com-

* Corresponding author. Email: dglee@de.etri.re.kr.

piler should be modeled so that its object model should reflect semantics of the language exactly. From semantic definitions of source and target language, elementary semantics is defined as a *class attribute*. A *class* is defined by grouping some class attributes with related class attributes; a *class operation* is defined as a function to evaluate class attributes in each class. As semantics in programming languages is defined well in language definition as well as in studies on compiler construction [1,7] based on attribute grammar, class attributes and class operations in the object model can be defined from those easily. Each class in the object model corresponds to each component of the program. The relation between classes is defined as an *association*. Since the association represents relation between classes, the attributes defined within the associated class are used through association. A *link*, which is an instance of an association, connects objects to each other. In the case of one-way association, a link can be implemented as a pointer (an attribute which refers an object) to other objects. The one end of traversing direction in one-way association is defined as the *destination* and the opposite as *source*. When objects are connected by a link, the object in the source is defined as the *source object*, and the object in destination is defined as the *destination object*.

As a compiler's object model intends to reflect program structure and its semantics exactly, the compiler can be considered as a set of functions evaluating language semantics represented in the object model. Therefore, it is possible to construct a compiler by collecting class operations defined in the object model. Moreover, calling sequences for these class operations become the dynamic behavior for the compiler. Accordingly, the compiler is modeled as four sequential phases: semantic initialization, semantic analysis, semantic transformation from the source language to the target language, and code generation. To describe dynamic behavior for compilers more easily, we assume the association as *one-way association*.

As an example, let us consider Pascal [3]. A Pascal program consists of three parts, program structure (PS), data object (DO), and action (ACT). Data objects are described by declaration and definition statement. Actions are described by means of action statement. We discuss compiler modeling only with these three parts and the relations among those with a brief explanation. Simplified object model for the Pascal is illustrated in Fig. 1.

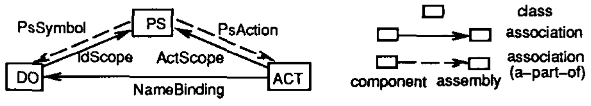

Fig. 1. Simplified object model for compiler.

3. State concepts for dynamic configuration

Since the compiler is a batch processing algorithm, its dynamic behavior can be represented by a *finite state machine* [2] which processes the program semantics. To represent how the program semantics is processed by the finite state machine, it is necessary to describe changes in values of class attributes and objects as states and transitions of the finite state machine.

The value of class attribute varies according to the class operations which evaluate on itself. The situation in which a class attribute has a particular value is defined as a *state* of a class attribute, and the change in value of a class attribute is defined as a *state transition* of the class attribute. Since the program semantics in compiler is processed through phases, the value of each class attribute varies accordingly. To describe the changes in values of class attributes in temporal order, we extend the class attribute and define it as the *attribute*. Due to the fact the the attribute is a dynamic representation of the class attribute (see below), this paper distinguishes "attribute" from "class attribute". Just as an attribute has its own state according to its value, each object also has its own state according to the values of all its class attributes.

Definition 3.1 (Attribute). An *attribute* is a triple $A = (N, C, S)$, where N is a class attribute, C is a class name including the N, and S is an attribute state. The attribute state is a positive integer, where 0 is the uninitialized state, 1 is the initialization state. The attribute state can be increased whenever the value of attribute is changed. We denote an attribute as $N(C, S)$.

Definition 3.2 (Object state and its definition). An *object state definition* is a triple $SD = (C, S, D)$, where C is a class name of an object, S is an *object state*, and D is a constructed domain by production of attributes. Let the C class have n class attributes, and d_i be an attribute for the ith class attribute where $1 \leqslant i \leqslant n$. Then $D = \langle d_1, d_2, \dots d_n \rangle$. Object state is a positive integer, where 0 is the uninitialized state, 1 is the initialization state. The object state can be increased whenever the value of the object is changed in time. We denote an object state definition as $C[S] = d_1 \times d_2 \times d_3 \times \cdots \times d_n$;

In Fig. 2, *name*(PS, 1) represents that *name* is a class attribute in the PS class and has its own initial value. Any desired object state can be defined following the keyword *STATE*. PS[1] indicates that it is the first state for an object of PS and the state is defined by *symbol*(PS, 1), *action*(PS, 1), *name*(PS, 1), and so on. Whenever some attributes in an object state have been evaluated, the state of the object is transmitted to the next state that is defined in the object state definition. Fig. 2 is a description of a dynamic behavior of the object model that is shown in Fig. 1. In Fig. 2, PS[1], PS[2] and PS[3] defined in PS module describe the object states that are generated when an object of PS is compiled. ACT[1] and DO[1], which are defined within ACT module and DO module, also describe the object states of object of ACT and object of DO, respectively. In this paper, we assume that each object has four states as follows: PS[1], ACT[1], and DO[1] are the states for semantic initialization; PS[2], ACT[2], and DO[2] are the states for semantic analysis; PS[3], ACT[3], and DO[3] are the states for semantic transformation; PS[4], ACT[4], and DO[4] are the states for code generation states.

```
FOR  PS  /* class name is PS */
MODULE
STATE  /* state definition */
    PS[1] = symbol(PS,1) x action(PS,1) x name(PS,1) ..... ;
    PS[2] = symbol(PS,2) x action(PS,2) x name(PS,1) ...... ;
    PS[3] = symbol(PS,3) x action(PS,3) x name(PS,1) ......;
              . . . . . . . . . . . . . .
CHANNEL  /* channel attribute definition */
    symbol = PsSymbol(.....) ..... FROM PS TO DO;
    action = PsAction(.....) ..... FROM PS TO ACT;
              . . . . . . . . . . . . . .
EQUATION  /* attribute equation */
    name(PS,1) := idString(PS,1) x blockId(PS,1),
                  genUniPsName;
    symbol(PS,1) := SYNCH(DO[1]), PsSymbol;
    symbol(PS,2) := CHANGE(DO[1] : DO[2]), DoAnalysis;
    symbol(PS,3) := CHANGE(DO[2] : DO[3]), DoTrans;
    action(PS,1) := SYNCH(ACT[1]), PsAction;
    action(PS,2) := CHANGE(ACT[1] : ACT[2]), ActAnalysis   ;
    action(PS,3) := CHANGE(ACT[2] : ACT[3]), ActTrans;
                 . . . . . . . . . . . . . .
END;
```

```
FOR  DO
MODULE
STATE
      DO[1] = ....... ;
       . . . . . . . . . .
 . . . . . . . . . .
END
```

```
FOR  ACT
MODULE
STATE
    ACT[1] = ......  ;
     . . . . . . . . . .
 . . . . . . . . . .
END;
```

Fig. 2. Equation modules of PS in Fig. 1.

As a link connects a source object and a destination object, it is possible for the source object to access the attributes in the destination object through the link. However, in the dynamic behavior, the states of linked objects are changed with the lapse of time so that the meaning of a link depends on the states of a source object and a destination object. To describe the changes in the state of a destination object, we define *link state* as the state of the destination object. We also define a *channel attribute* as the attribute representing the link state.

Definition 3.3 (Channel attribute and its definition). A *channel attribute definition* is a $CA = (A, AS, S, D)$, where A is the name of a *channel attribute*, AS is a association for a link, S is a source class of AS, and D is a destination class of AS. We denote a channel attribute definition as $A = AS$ FROM S TO D;

In Fig. 2, channel attributes are defined after the keyword *CHANNEL*. In Fig. 1, associations whose sources are PS are *PsSymbol* and *PsAction*. So, channel attributes for these associations are defined as *symbol* and *action*. Both *symbol* and *action* are pseudo class attributes of PS. An association includes the name of association, source, destination and so on. In the channel attribute, *action*(PS, 2), belongs to the state PS[2], and the state of the destination object is ACT[2]. Therefore, *action*(PS, 2) indicates that it is a link which connects an object in *PS*[2] state with an object in *ACT*[2] state.

When attempting to use attributes within the destination object, if a destination object does not reach the desired state which is specified by a channel attribute, it is impossible to access the destination object through the link. Therefore access to the destination object has to be delayed until the destination object reaches the desired state as specified by the channel attribute.

The attribute, object state and channel attribute are *dynamic representations* of the class attribute, objects and link respectively. They are considered as basic concepts representing dynamic behavior of the compiler.

4. Attribute equations

We describe dynamic behavior of objects with an *equation module* for each corresponding to a class. Each equation compiler consists of object state definition, channel attribute definition, precedence definition, and attribute equation parts. The object state definitions define the object states to be synchronized or to be activated from themselves or other objects via links among several object states through which an object goes during compiling the object. The channel attribute definitions channel attributes with qualified one-way association. The *precedence* may be used only when evaluation order is determined by an *external directive* not by semantics dependency between attributes. For example, in code generation phase, the precedence is useful in determining evaluation order of attributes, since the code generation sequence is determined by code patterns of the target language. Attribute equations describe dynamic dependency between attributes and between objects. Forms of attribute equations are defined in three types as follows:

Definition 4.1 (Type 1: attribute equation). A *type*-1 *attribute equation* is $E=(A, R, F, S)$ where F is a function defined as a class operation in an object model, A is an attribute for the output value of F, R is a relation "evaluated-by" that is transitively, and S is a set of attributes for input arguments of F. The attribute equation is denoted as $A := s_1 \times s_2 \times \cdots \times s_n$, F where s_i is in S and $1 \leqslant i \leqslant n$. We say that A is evaluated by $(s_1, s_2, \ldots, s_n)$ with F. Note that a channel attribute as an input argument for F should not be used in a type 1 equation. However, a channel attribute can be used only if it is used as a link to access attributes defined in the other objects.

Definition 4.2 (Type 2: attribute equation). A *type*-2 *attribute equation* is $E=(A, R, F, S)$ where A is a channel attribute, R is a relation "evaluated-by" that is transitively closed, and S is the SYNCH function which takes the state of the destination object as its argument. The SYNCH function sets the state of a destination object to A so that the source object cannot use the attributes of the destination object through link until the destination object reaches specified state by the argument. F is a function to set-up a link between related objects by association and takes name of association name as its name. The attribute equation is denoted as

$$A := \mathit{SYNCH}(\text{objectState}),\ F.$$

Definition 4.3 (Type 3: attribute equation). A *type*-3 *attribute equation* is $E=(A, R, F, S)$ where A is a channel attribute, R is a relation "evaluated-by" that is transitively closed, and S is the CHANGE function. The CHANGE function uses both the starting and ending state of transition as its arguments. The CHANGE function activates the destination object from the starting state to the ending state by executing class operations that needed for the state transitions. Finally the CHANGE function sets the ending state of the transition to A and then, the source object can access the destination object. F is a driver routine for all functions to be executed during state transitions of the destination object. The attribute equation is denoted as

$$A := \mathit{CHANGE}(\text{startState:endState}),\ F.$$

In Fig. 2, the equation that evaluates *name*(*PS*, 1) is a type-1 equation. *idString* is an identifier of a program structure, whereas *blockId* is a unique block number of the program structure. *genUniPsName*, which takes *idString*(*PS*, 1) and *blockId*(*PS*, 1) as its input, is a class operation to generate a unique name even if several program structures have the same *idString* in a program, yielding *name*(*PS*, 1). In this manner, a type-1 equation represents semantic dependency between input argument attributes and resulting attribute of a class operation. The equation which evaluates *symbol*(*PS*, 1) is a type-2 equation. *DO*[1] indicates the first object state of DO, while *PsSymbol* is not only an association name but also a name of function which connects an object of PS with an object of DO by a link. Even if an object of PS can connect with an object of DO through a link, the object of PS can not access through *symbol*(*PS*, 1) until the object of DO reaches *DO*[1] state. The equation which evaluates *symbol*(*PS*, 2) is a type-3 equation. This equation enables an object of DO to activate all operations required to transit the object state from *DO*[1] to *DO*[2]. A function name, *DoAnalysis*, is the name of driver routine which invokes class operations in DO class and is executed by the CHANGE function.

In designing object model for compiler, each class corresponds to a component in a program, and a program is constructed by all the components. Thus, all associations defined in the object model are classified into two kinds of associations. One is *a-part-of relationship* in which objects representing the *components* of thing are associated with an object representing the *assembly*. The other is association existing between components in a-part-of relationship. Looking at the relations between objects during compiling, an assembly object should compile its component objects since compiling of the assembly object can be done only when both the assembly object and its component objects are compiled completely. Also, a component object can use attributes evaluated either in the assembly object or in the other component objects. By using defined object states, the modeling of dynamic behavior for compilers is described with object synchronization and object activation.

The dynamic behavior of an object model for compilers is described by using attribute equations as follows:

1. In an assembly object, a type-3 equation describes that the assembly object activates component objects to the desired state to evaluate its component objects.
2. In a component object, a type-2 equation describes an object synchronization with its assembly object so that the component object can access to the attributes in the assembly object.
3. Between component objects, a type-2 equation describes an object synchronization for the source object to access to the attributes in the destination object.
4. Within an object, a type-1 equation describes attribute evaluation in every object state.

Therefore, the dynamic behavior for compilers is described with three kinds of attribute equations.

Fig. 3 shows a skeleton program for *PS* in Fig. 2. To generate a skeleton program for *PS*, a driver function for an object state should be defined so that all the class operations in an object state can be executed by the topological order. The driver function is called as a *state function*. For example, *ps1* in Fig. 3 is a state function for *PS*[1]. By the definitions of attribute equations, every attribute equations is translated into the code for calling its function that is specified in the attribute equation. As an example, in Fig. 3, *PsSymbol*() defined in type-2 equation is a function name which establishes the link of *PsSymbol* association. *DoAnalysis*() defined in

```
ps1() { ....; genUniPsName(); ....; PsSymbol(); ...., PsActiion(); ..... ; }
ps2() { .....; DoAnalysis(); .....; ActAnalysis(); .....; }
ps3() { .....; DoTrans(); .....; ActTrans(); .....; }
   ......  ...........

ps() { ps1(); ps2(); ps3(); ......  }
```

```
DoAnalysis() { do2() }
DoTrans() { do3() }
 ......  ...........
```

```
ActAnalysis() { act2() }
ActTrans() {act3() }
 ......  ...........
```

Fig. 3. Skeleton compiler program for Fig. 2.

type-3 equation is a driver function that calls state function to change the object state of *DO*. In addition, main procedure, *ps*(), is generated to call all the state functions for objects of root assembly class in the model by the topological order.

5. Evaluation of attribute equations

The fundamental steps to generate control structure of a compiler from attribute equations are as follows:

(1) Represent an object state defined in each equation module as a vertex. Analyze CHANGE function and SYNCH function to define relationship between object states, and then represent its predecessor/successor relation between object states as an edge to produce a directed graph. From the produced directed graph, determine a transition order of all object states by the topological ordering.

(2) Determine the evaluation order of operations performed within an object state. Define the predecessor/successor relations between attributes by using the following conditions:

(a) Define the predecessor/successor relations of attributes from the definition of object state.

(b) Define the predecessor/successor relations of attributes between input argument attributes and result attribute of class operations.

(c) Determine the predecessor/successor relations of channel attributes by using the topological order of an object state, if several channel attributes are defined in the same object state and they are links of different associations.

(d) For attributes with no semantic dependency, predecessor/successor relations can be determined by the precedence given by the external directives.

We represent a predecessor/successor relation determined by above conditions as an edge, and represent both an attribute and a class operation to evaluate the attribute as a vertex so that a directed graph can be produced. From the topological ordering of the produced directed graph, the evaluation order of attributes, which is the calling sequence of the class operations, is determined. Furthermore, the dynamic behavior for operations in one object is determined according to the above procedures.

(3) It is decidable to detect a cycle from the directed graph produced by step 1 or step 2, and the detected cycle can be removed by splitting object states or attribute states as follows:

(a) The dependency among attributes can cause circularity in the directed graph. Compiling of type declarations introduce apparent circularities of attributes. As an example, **type** *t* = **record** *x*: **real**; *p*: **ref** *t* **end**; in Pascal, the names should be resolved in the order of *t*, *x*, *p*, and *t* to evaluate the attributes of *t*. However, as shown in Fig. 4, the circularity can be resolved at run-time by the operations that detect and remove the cycle. First, split two states *p* and *q*, which cause a cycle, into two continuous states. That is, split *p* into *p1* and *p2*, and split *q* into *q1* and *q2* states. *p1* and *q1* are starting states of the recursion and *p2* and *q2* are ending states of the recursion. At this point, define states so that the transition order of states should be as *p1*, *q1*, *p2*, and *q2*. Secondly, connect all in-edges of *p* with in-edge of *p1*, all out-edges of *p* with out-edges of *p2*.

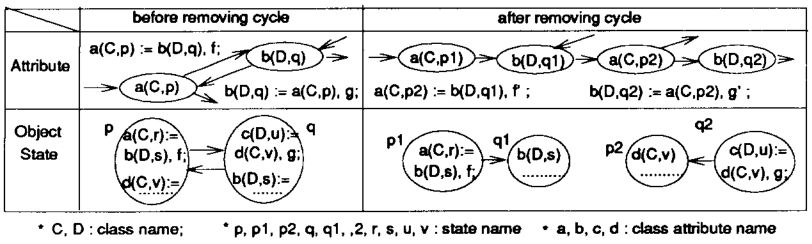

Fig. 4. Cycle removing in attributes and object states.

Similarly, connect all in-edges of q with in-edge of $q1$, all out-edges of q with out-edge of $q2$. Each of class operation in p and q is assigned to the ending states, $p2$ and $q2$, respectively. If there are some states between p and q and there is no recursion, they also should be split in the same manner. Finally, a facility to detect and remove the recursion in run-time should be added to the operations because this method is applied for only determining the topological order of class operations under the assumption that the cycle is removed by the class operations during run-time.

(b) The cycles can be also caused by the definitions of object states. Although there is no attribute dependency that cause cycle among object states, if several attribute dependencies exist between object states as shown in Fig. 4, a cycle can be generated by the combination of some attribute dependencies. In this case, first, split states on the cycle so that the cycle should be removed through redefining object states. Secondly, to connect with new states that have the desired attributes, rewrite attribute equations in both states to be split and states to be connected with new states through in-edges or out-edges. Finally, iterate from step 1 to step 3.

6. Discussions

In this paper, by suggesting a description method for the dynamic behavior of class operations defined in an object model for compilers, we can conclude that the compiler description based on object-oriented modeling is a feasible method to improve modularity of compilers for complex high-level programming languages. A more generalized description method will be discussed in detail in [4], which supports *class generation* in the object model. The result of implementation of attribute equations for a Pascal subset with almost of all the full Pascal features is shown in Table 1.

Table 1
Experimental result

No. of classes	31
No. of attributes	103
No. of states	93
No. of associations	61
No. of precedences	23
No. of equations	324

With the formal basis for translations in classical compiler, Reiss [5] defined a compiler model that consists of several phases. The semantics used in each phases is defined as objects. The translations to be applied to the objects are also defined as primitive functions. The modules corresponding to each phase are generated from the specifications that describe objects and interactions with the syntax and semantics. This approach can support data abstraction for modules such as symbol processing mechanisms. However semantic analysis is done by an attributed grammar evaluation of the abstract syntax tree.

By treating grammar symbols as a class, Wu and Wang [8] defined class hierarchy based on syntax grammar so that the nodes of parse tree are treated as the objects of their classes. They

Table 2
Comparison of attribute grammar and attribute equation

	attribute grammar	attribute equation
Elemently semantics	attribute of syntax grammar	class attribute in object model
Relationship of semantics	grammar symbol	association
Specifications method	attribute grammar	object model for language semantics, attribute equation
Attribute propagation	propagation on parsing tree, prespecified order defined based on the graph	propagation on the association by the link
Evalution order	tree traverse order	topological order from attribute equations

defined the class hierarchy with inheritance among a production symbol and the symbols in its right-hand side. Besides, in order to specify the paths to receiver nodes of value from its sender, they presented a method to describe the path information from the class hierarchy and structured parse tree.

Both Reiss [5] and Wu and Wang [8] used object-oriented concepts partially in supporting the data abstraction of compiler modules and describing the path expression based on class hierarchy, respectively. However, these approaches are analogous to the attribute grammar approach in fact that they represented semantics on the syntax and evaluated the semantics on the abstract syntax tree. Our approach make easy to understand the specifications since it is based on the object model of language semantics. Moreover, our approach can reduce difficulties in describing evaluation of complicated semantics since it describes evaluation of semantics with the attribute equations based on associations. Table 2 compares characteristics of our approach with attribute grammars [1,7] as a specification method for compilers.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, New York, 1986).

[2] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, MA, 1979).

[3] K. Jensen and N. Wirth, *Pascal User Manual and Report* (Springer, New York, 1978).

[4] D. Lee, A specification of compilers based on the object-oriented paradigms, Ph.D. Thesis, Dept. of Computer Science, KAIST, South Korea, in preparation.

[5] S.P. Reiss, Automatic compiler production: The front end, *IEEE Trans. Software Engrg.* **13** (6) (1987) 609–627.

[6] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design* (Prentice-Hall, Englewood Cliffs, NJ, 1991).

[7] W.M. Waite and G. Goos, *Compiler Construction* (Springer, Berlin, 1984).

[8] P.-C. Wu and F.-J. Wang, An object-oriented specification for compiler, *ACM SIGPLAN Notices* **27** (1) (1992) 85–94.