データベース・システム 79－6
プログラミング言語 26－6
（1990. 9. 7）

属性文法に基づいたオブジェクト管理システムOOAGにおける構造進化

譚 立朝　篠田 陽一　片山 卓也

東京工業大学工学部情報工学科

本文は属性文法に基づいたオブジェクト管理モデルOOAGのプロトタイプシステムOS/0における構造的な変化とその処理機構について述べる。OOAGはオブジェクト指向的なモデルであって、木構造を持ったドメインへの記述に特に優れている。このシステムで許されるオブジェクトの構造変化は木のタイプの進化と木インスタンスの内部構造の動的な変更などがある。これらの動的変化はメタ・オブジェクトで特徴付けられたメカニズムによって制御され評価される。また、本文は本システムのソフトウェア・プロセスにおける変化への対応についても議論する。

Attribute Grammar Based Structure Evolution in Object Management System OOAG

Lichao Tan　Yoichi Shinoda　Takuya Katayama

Department of Computer Science, Tokyo Institute of Technology
2-12-1 Ookayama Meguro-ku Tokyo 152 Japan

In this paper, we discuss methods of dealing with structural changes in an object management system OS/0, which is a prototype of an attribute grammar based object management model OOAG. OOAG is object-oriented and especially suitable for describing tree-like structures that are widely used in practice. Structural changes in the system include the evolution of tree type and dynamic transformation of tree instance's internal structure. The changes are controlled and evaluated by an efficient mechanism that is characterized by *meta-object*. Dealing with change in the context of software process is also discussed.

1 Introduction

Motivation for improving a software development environment (SDE) may derive from three different requirements for improving used tools, software processes and software artifact structures. Requirements for improvement in tools may derive from two different aspects: deficiencies in current tools and new requirements on the part of the user. Both cases may result in evolution of current tools or creation of new tools. However, a good new tool would lose much of its effectiveness if it is incompatible with the old ones. So, it is generally desirable that the user's view of a system be kept consistent as much as possible when the system changes. On the other hand, for a modern SDE, support for improvements of software processes, as well as their associated artifacts is considered indispensable. Another problem associated with change in a SDE is how to deal with the old states (tools, structures, views, etc) of the environment. It is convenient and necessary to save and manage these old states for certain software development activities such as system re-adjustment, reproduction of certain states, re-testing of old products and so on. There is also a problem of interface for user because of the need to access old states or to shift between new and old states in an efficient way.

In this paper, we discuss the method of dealing with these problems in an object management system OS/0 (*Object System / Zero*) whose model is based on a formalism called OOAG (*Object-Oriented Attribute Grammars*)[SK 90a]. OOAG is an object-oriented extension of standard attribute grammars [Knuth 68] that combines the advantages of both declarative (functional) and object-oriented paradigms. Substantial changes in the system include dynamic transformation of object structure as well as dynamic modification of object class definition, which collectively provide to support the improvement of SDE as mentioned above. In OS/0, these changes are controlled by *meta-objects* in order to provide a clear and consistent user view. Old-fashioned objects are identified as *versions* of current objects and are also managed by the meta-object mechanism.

The paper is organized as follows. In section 2, we first explain the OOAG model. Then, in section 3 we elucidate how structural change of objects can occur in the model. In section 4, we will describe how software processes and changes in them are described using the OOAG model. A method for controlling and evaluating the changes is discussed in section 5; in this section, the functionalities of meta-object, the main element of the method, are described and a kind of lazy evaluation of changes by meta-object is also discussed. Section 6 discusses some implementation issues. Finally, section 7 gives some concluding remarks.

2 The Principle of OOAG

Traditionally, *attribute grammars* are known for their use in generating compilers for programming languages. However, because attribute grammars are based on the functional computation of attribute values over the tree structures, it can be used as a powerful mechanism wherever the target application can be described as a system of tree structures with operations manipulating values called *attributes* attached to their nodes[Kat 89]. Object-Oriented Attribute Grammars (OOAG) has been proposed mainly for managing software objects in software development environments[SK 90a]. OOAG incorporates certain object-oriented features into standard attribute grammars in order to solve problems inherent in standard attribute grammars. Among these problems are the following:

1. Since the meaning of an entire attributed tree is given only by the values of distinguished attributes of the root node, any results from significant semantic computation must be brought up to the root.

2. There is no mechanism to dynamically invoke recalculation of attributes, which is necessary for practical applications.

3. Because every attribute is permanently attached to a node, even temporary attributes must be still described and their values maintained in the same way as other attributes.

4. Modification of a tree including creation of new nodes is not possible from inside the grammar, i.e., as a standard computational mechanism offered by semantic functions.

In OOAG, attributed trees are viewed as distinguished *objects* capable of receiving and understanding *messages*. Object definition is performed in two parts: *static*

specification and *dynamic specification*. The static specification is compatible with standard attribute grammars. It specifies the structure of the object (syntactic rules) and static semantic rules. Static semantic rules are used to express relations that have to be maintained among *static* attributes and *native* attributes. Native attributes are values that express the nature of the object and are permanently attached to the object. Static attributes are divided into two kinds: *inherited attributes* and *synthesized attributes*; both are used to propagate attribute values over the tree. For example, consider a program P that is divided into two module objects M_1 and M_2 whose source programs are written in C, or divided into three module objects N_1, N_2 and N_3 whose source programs are written in Fortran. The executable code for P must be updated whenever the object code of any module is modified. The specification of such a P can be described as below. This specification is explained further in the following subsections.

```
P( spec | executable ) → M1 M2
    - - -static specification
    where                  --static semantic rules
        M1.spec = specPart1( P.spec )
        M2.spec = specPart2( P.spec )
        P.executable = link(M1.obj_code, M2.obj_code)

    - - -dynamic specification
    :modifySpec ( specDelta | ) ⇒
        M1 :modifySpec( specDelta1 | )
        M2 :modifySpec( specDelta2 | )
    where          --dynamic semantic rules
        specDelta1 = specPart1( specDelta )
        specDelta2 = specPart2( specDelta )
    :
    :
    → N1 N2 N3          - - -another static specification
    where
        :
        P.executable =
            link( N1.obj_code, N2.obj_code, N3.obj_code)
        :
```

2.1 Static and Dynamic Specification

The static specification for object P above shows that P has a static inherited attribute *spec*, a static synthesized attribute *executable*, and two component objects M_1 and M_2 (in the first case). Below the *construction rule* $P \rightarrow M_1\ M_2$, which corresponds to the syntactic rule in standard attribute grammars, synthesized attributes (output) of P and inherited attributes (input) of M_1 and M_2 are defined by the static semantic rules.

A dynamic specification has two basic components: an *activation rule* and some *dynamic semantic rules*. An activation rule is of the form : $message(i_1...i_p \mid o_1...o_q) \Rightarrow R_1 : m_1 \cdots R_l : m_l$, where $i_1,..,i_p$ and $o_1,..,o_q$ are *dynamic input(inherited)* and *output(synthesized)* attributes respectively, and $m_1,...,m_l$ are messages to the components $R_1,...,R_l$. A dynamic semantic rule is similar to a static semantic rule except that it can define and use the value of native attributes and the dynamic input/output attributes. Dynamic attributes are somewhat different than static attributes. Specifically, static attributes are persistent in the tree, while dynamic ones become alive (visible) only when the activation rule in which they occur is invoked by message. Moreover when the computation is finished, they disappear again. In other words, dynamic attributes can be viewed as temporary attributes used for dynamic computation. The example also shows the definition of a :modifySpec message which asks for the modification of the program's specification. We will explain later how this message results in modification of the program's executable code.

2.2 LHS and RHS Classes

When we take an object-oriented view of a construction rule, the *left hand side (lhs)* symbol can be viewed as a class name while the *right hand side (rhs)* part as giving the class definition. Expanding the *lhs* node to its *rhs* nodes in a tree is viewed as creating an instance of the class. We note that there are two construction rules for the same object P in the example, that is, there are two possible ways to expand the node P in a tree. Applying either construction rule creates a valid instance of P. Hence, we consider the *rhs* part of each alternative construction rule as giving the definition for a subgroup of objects of the LHS class represented by the *lhs* symbol. Such a subgroup is called as a RHS class, and the LHS class is regarded as the union of the alternative RHS classes. RHS class names may be explicitly specified in the static specification by a prefix dot notation, e.g.,

P(spec | executable) $\rightarrow$ .C M_1 M_2

:

$\rightarrow$ .F77 N_1 N_2 N_3

:

In this case, the LHS class P is regarded as the union of two RHS classes .C and .F77 (technically, RHSs are made to inherit the LHS). A LHS class actually has its own definition of instance and class methods below the RHS definition part, but we omit them here for convenience.

2.3 Dynamic Computation

Dynamic computation in an object is invoked by sending a message to the object with appropriate values bound to its dynamic input attributes. On receiving the message, the object selects the corresponding dynamic specification and evaluates its dynamic semantic rules by (1) evaluating the equations and/or (2) sending appropriate messages to objects that appear in the right hand side of the activation rule. Component objects receiving these messages repeat the same action, potentially leading to further computation and further messages. After this dynamic computation has finished, the object modifies its static attributes by re-evaluating the related static semantic rules if values of its native attributes on which the static attributes depend have been modified. The modification of its static attributes in turn causes an attribute change propagation through the tree, thus bringing the entire tree to a new consistent state.

Consider the following specification of the "module" object M used in the preceding example.

```
M ( spec | obj_code ) → R      − − −static spec
    where
        R.spec = M.spec
        M.obj_code = cc ( R.source )
    native current_spec, current_obj
    where
        M.obj_code = current_obj

    − − −dynamic specification
    :modifySpec ( specDelta | ) ⇒
        R :makeSource (
                (new current_spec) | current_source )
    where
        (new current_spec) =
                makeSpec ( current_spec, specDelta )
        current_obj = cc ( current_source )
    :
```

In the static specification part, it specifies that module M has a component object R that constructs a source program from its specification. It also specifies two native attributes, current_spec and current_obj, which store the newest specification and object code of the module. These two native attributes are modified by the message :modifySpec. When the newest object code current_obj is obtained, the static synthesized attribute obj_code of M is reset to the new current_obj by the static semantic rule. The value of obj_code is in turn propagated to the P object affecting the attribute executable. The notation (new ...) is used to represent the new value of native attributes.

Having explained the general features of OOAG, we now turn to discussing those aspects of the system that are designed to cope with change.

3 Dealing With Change in OOAG

Upon receiving a message, an object may modify the values of its native attributes as a result of the corresponding computation. We consider this kind of change trivial from the point of view of system evolution. Rather, What we discuss in this paper are structural changes, which cause more significant problems with respect to preserving a consistent view of the system. In OOAG, there are roughly two kinds of structural change by now. One is in the instance level; an object can transform its internal structure from one RHS to another legal RHS. The other one is caused by changes to the LHS/RHS class definition, which again affects their related instance objects. Another aspect of dealing with object change is how to provide an efficient tool for manipulating the change history. In this section we discuss how all of these problems have been addressed in OOAG.

3.1 Structural Transformation of Objects

As stated in section 2, the definition of a LHS class consists of several alternative RHS definitions. Instances associated with any of these classes are regarded as valid instances of the LHS. A *transform* operation converts

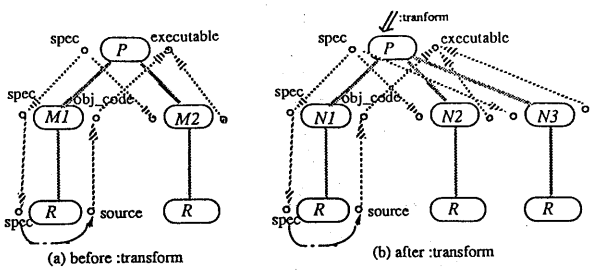

Figure 1: Object State Before and After :transform

an instance of a RHS into an instance of another alternative RHS class. Transform operations are invoked by the message :transform, which is realized in the LHS class. After the transformation, the object's attributes are re-computed or reset properly and the propagation of static attributes is carried out according to the appropriate semantic rules.

Figure 1 shows the state of P in the example before and after a :transform operation. Note, however, that the interface of P does not change so that the user can send the same message to P before and after the transformation (e.g., sending the same :exec_code returns the old *P*.executable (C-code) before and the new one (F77-code) after). Keeping the interface of the LHS class unaffected in this way provides a consistent object view to the user, who see objects only in terms of their LHS. We restrict the means of instance conversion to explicit invocation by the message :transform. This is in contrast to the method of allowing arbitrary user-defined object conversion operations. As we only allow predefined conversion rules, that is, only those RHS classes corresponding to the LHS are legal candidates for transformation, the semantic correctness of the transformation is guaranteed. In managing large objects, this kind of restricted transformational ability is considered both efficient and essential since arbitrary transformations may break the system's consistency. Moreover, it is often difficult to define arbitrary transformation between objects in any situation.

3.2 Changes to Class Definition

In OOAG, a LHS or RHS class can dynamically change its own definition by receiving messages. Currently there are three basic types of class definition change.

1. A new RHS definition is added to a LHS class. This operation does not affect current instances, but the new RHS can be used later for object instantiation or as a new candidate for a :transform.

2. An existing RHS definition is modified in a LHS class. This operation results in the invocation of :transform operation to all the current instances of the RHS in order to keep them consistent with the new definition.

3. An existing RHS definition is deleted in a LHS class. This operation also causes the :transform operations on the instances of the RHS with the effect of converting them into UNDEFINED objects.

These three operations are realized in the LHS class by the class methods :addRhs, :modifyRhs, and :deleteRhs, respectively. Other more complex changes can be realized by using them, e.g., deleting a LHS wholly can be resolved to deleting all the RHS definitions in the LHS. We address the problem of how to maintain old instances caused by such class evolution in the following section.

3.3 Version Mechanism for Change History

In this section, we discuss the version mechanism we are adopting in OOAG. It is through the mechanism that users can access the old structures of objects and perform activities such as retrying, testing and comparing. By the benefits of static properties offered by the attribute grammar, we can easily move to the correct old state of a composite object by attribute propagation when one component object backtracks to an old version.

A new version of an object is created in two ways. One is explicitly by a message :checkIn. When an object receives a :checkIn message, its current state is saved as a new stable version of the object. A user may use the :checkIn message at any time. The other is implicitly when the object receives a :transform message and performs the appropriate operation. The state before the

transformation is logged as a new stable version, while the state after the transformation becomes the current state. Thus, a new version is made as a side effect of performing each :transform.

In OOAG, a message is sent by default to the current version of the object. As a result, accessing an old version requires two steps. First we need to *pop up* an old version, making it the current version. This means that subsequent messages will go to this version. Second, we send messages to the object to perform the actual access. A set of version pop-up operations are defined as instance methods of RHS classes such as, :popUpNo(aNum), :popUpNewest(), :popUpSucc(), :popUpPred(), and :popUpRhs(aRHS). The message :popUpNo(aNum) requests that version aNum be popped up as the current version. The message :popUpRhs(aRHS) requests that one version which is an instance of the aRHS class be popped up; when there are many such instances, one is chosen according to a default rule. The :popUpNewest(), :popUpSucc() and :popUpPred() messages request that the newest (in time) version, the successor, and the predecessor of the current version be popped up, respectively.

We note that only version creation operations (:checkIn, :transform) or pop-up operations can result in new current versions. It is possible to get a non-linear version derivation graph by, for example, applying a version creation operation to an old version (by popping it up first). Consider now the problem of dealing with old versions when a class is changed, e.g., "should we delete all the old instances when a RHS class is deleted?". There are obviously many possible solutions. Our solution is that changes in the current class definition do not affect the old instances, i.e., when a RHS is modified or deleted, the old versions of its instances are not touched even though the current versions may change. We consider this may reduce the cost of class change, yet still be convenient for referring to real change histories.

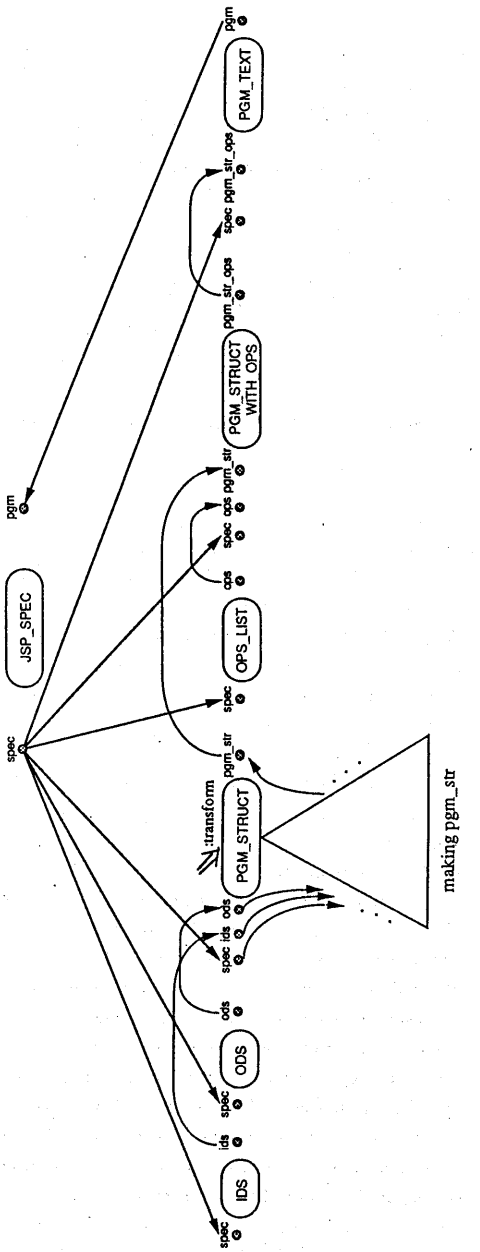

Figure 2: Graphical Representation of JSP Object Specification

4 Change Support for Software Processes

4.1 The Object-Oriented Software Processes

One important aspect of describing software objects in the OOAG formalism is that data driven software processes and their enaction can be described naturally. Software processes are viewed as hierarchies of software activities, each of which corresponds to a node in the tree. An initial software process is enacted by providing the initial inherited attributes to a newly created tree of software objects and proceeds as the static computation is carried out. Re-adjusting a software process, including dynamic structure modification and external tools exchange, can be realized by change invoking messages.

As an example, consider the Jackson Structured Programming (JSP) method[Jac 75]. JSP is a structured method of producing programs that can be informally described as consisting of the following steps:

1. Construct data structures: the specification is analyzed and two data structures called the *input data structure* and the *output data structure* are extracted.

2. Derive program structure: the input and output data structures are put together to derive the program structure.

3. Enumerate operations: the specification is analyzed, and operations to be performed on input and output data are extracted.

4. Create programs: the program structure obtained from step 2 is augmented with operations from step 3 to form a program structure with operations. The final program is then derived from this structure.

To describe this method using OOAG, we first extract seven software artifacts as distinguished objects: (1) the specification (JSP_SPEC) (2) the input data structure (IDS) (3) the output data structure (ODS) (4) the program structure (PGM_STRUCT) (5) the operation list (OPS_LIST) (6) the program structure with operations (PGM_STRUCT_WITH_OPS) and (7) the final program text (PGM_TEXT). Next, we distinguish the attributes to be attached to each of these objects as well as semantic dependencies among these attributes. Resulting attributes and semantic dependencies among them are shown in Figure 2.

Enaction of this JSP process is initiated by instantiating

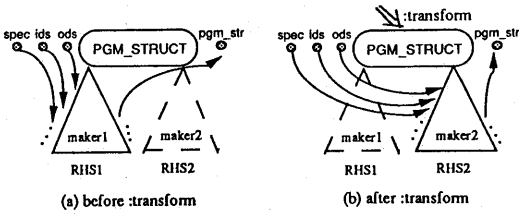

Figure 3: States of Object PGM_STRUCT

a JSP_SPEC object along with its children objects, and giving appropriate value to its static inherited attribute spec. This will propagate through the component objects, and the process will terminate with synthesized attribute pgm of the object JSP_SPEC properly computed.

Above described methodology of describing a software process can be called as *object-oriented software processes*, as analogy to the object-oriented design methodology. Various aspects of changes in software processes, including changes to related artifacts and tools used, can be described well in our OOAG based object-oriented software process framework. using change coping mechanisms described in earlier sections. In the followings, we show more details of how changes can be supported in the object-oriented software processes.

4.2 Coping With A Simple Change

Suppose now that the object PGM_STRUCT in the example is equipped with two alternative methods for constructing the program structure pgm_str from inputs spec, ids and ods. Also assume that when the JSP process proceeds to some point and the results are unsatisfactory, it comes out the method used in PGM_STRUCT was not proper and, hence, needs to be shifted to the other one. This change requirement can be satisfied by sending the message :transform to PGM_STRUCT (Figure 2). The states of PGM_STRUCT before and after the :transform are illustrated in Figure 3. As discussed in section 2, we can regard the LHS class PGM_STRUCT

as having two alternative RHS classes. When we take into account the versioning of products created during the process, all the related products (represented as attributes of the nodes appearing to the right of PGM_STRUCT in Figure 2) created before the invocation of the :transform are saved as old versions. An entire collection of versions from a given time comprise the product list of the whole process at that time. Such a collection can be obtained by sending a special version pop-up message to the root; and as a result, the root will in turn send pop-up messages to its subtrees to gather appropriate component products. As shown in this example, structural changes can be invoked and responded to well during software processes.

5 Monitoring Evaluation of Change by Meta-Object

One of our major objectives in this work has been to provide a consistent view of objects so that users can freely manipulate the old versions without the need for usage modification. Moreover, we also wanted to make it efficient to evaluate validness-guaranteed object structural changes. To realize these objectives, we use *meta-objects* for version control and change evaluation monitoring.

5.1 Meta-Object

The concept of meta-object was developed to represent computational reflection in object-oriented systems [Maes 87]. It provides a flexible way for dynamically changing the system itself. The concept of *persistent meta-object* was introduced to realize schema evolution management in object-oriented databases [TK 89]. Here, we incorporate a similar technique into the OS/0 system to realize a consistent user view of the changing system.

A meta-object is associated with an object that it manages. The meta-object holds the change history and other evolution information of the object. Any access to the object is intercepted by the meta-object, so that it can complete various management tasks for the objects. For examples, operations of version reference can be realized inside the meta-object by manipulating the change history, and operations of invoking changes are catched by the meta-object so that their execution can be monitored easily. A meta-object conceptually lives as long as its associated object does. In OS/0, a meta-object is not directly visible to the user. As a result, the user does not need to define the meta-object for an object, but rather the system will do so automatically. In other words, the meta-object can be regarded as a virtual object at the user level. It is embodied at a lower level by an XLISP object. XLISP[Betz 85] language is used to implement OS/0 (see in section 6).

5.2 Realization of Version Control

The task of version control in OS/0 is completed by meta-objects. In OS/0, we don't mean all the objects need versioning; indeed, some classes of objects such as integers do not even have proper semantics for versioning. So we distinguish two classes of objects: VERSIONABLE-NODE and UNVERSIONABLE-NODE. These two classes inherit the common NODE class that specifies the common structure and behavior of objects (nodes). A versionable RHS may have both *versioned* or *non-versioned* instances. A versioned instance may be dynamically changed to a non-versioned one by a message, and vice versa. In contrast, an unversionable RHS is allowed to have non-versioned instances only. When a versioned object is changed (e.g., by :transform), the old state is saved as a version. However, if the change occurs in a non-versioned object, the old state is not saved.

In OS/0, versioned objects are associated with meta-objects. Such a meta-object is created in two situations (1) when a versioned object is created, or (2) when a non-versioned object is changed to a versioned object. Similarly, a meta-object is deleted (1) when the corresponding versioned object is deleted, or (2) when the versioned object is changed to a non-versioned object.

As stated in section 5.1, any message to a versioned object is intercepted by its meta-object. Upon interception, the meta-object does the corresponding version control tasks according to the type of message. More precisely, it does the following

1. :checkIn,
   Saves the current state of the associated object as a stable version, and modifies the current version number and the internally stored version list.
2. :transform,
   Saves the current state of the object, then sends

the current version the message :transform to perform the transformation; also modifies the current version and the version list.

3. *pop-up* messages,
   Modifies the current version number.

4. other messages,
   Forwards the message to the current version.

Meta-objects in OS/0 are implemented by using XLISP objects. To define various types of meta-objects, class inheritance can be used to form a class definition hierarchy for meta-objects.

5.3 Lazy Evaluation of Change

Besides to realize version control, meta-objects is also used to facilitate a kind of *lazy evaluation* of structural changes. As stated in previous sections, when a structural change occurs, related attribute values are propagated through the whole system to make it globally consistent. This propagation can be computational intensive if there are a large number of objects. Sometimes it is efficient to delay the propagation until its effects are needed. For example, structural change to an object may not be evaluated until the object is to be accessed by attribute propagation or message passing. This control task can easily be implemented by the associated meta-object since changes to an object are stored there, and since every access to the object is intercepted by the meta-object. By this access interception mechanism, the timing of evaluation invocation can be set properly and the eventual consistent state of the whole system is guaranteed.

6 Implementation Issues

The OS/0 system is composed of two parts. One is the system kernel and user defined object space based on the XLISP system, and the other is the editor/translator generated by the Cornell Synthesizer Generator [RT 89] system, which translates OOAG specifications into executable xlisp programs. Two major components of the system kernel are (1) the definition of the NODE class, which implements functions common to all objects such as the attribute evaluation algorithm and subtree manipulation, and (2) the object scheduler which implements context switching, concurrent execution of multiple objects, etc. As for change management, there are several important points. When a :new is sent to a VERSIONABLE-NODE in OOAG, two :new messages are invoked at the XLISP level: one is sent to the corresponding class counterpart, while the other is sent to the class META to create a meta-object for the new object. At the XLISP level, an object of class VERSIONABLE-NODE is implemented as a set of versions, each of which has an attribute meta referring to the common associated meta-object. A meta-object has some attributes such as current-v, version-list, etc. To realize message interception by meta-objects, each message-passing operation in OOAG

```
x: aMes
```

is translated into a XLISP *send* function

```
(send (send x :meta) :aMes)
```

The message :meta returns the meta-object of the receiver object.

7 Concluding Remarks

We have discussed how "change coping" mechanisms in Object Oriented Attribute Grammars can be used to support changes required for improving SDEs. Our approach is characterized by the powerful mechanisms to cope with dynamic changes of type definition as well as internal structure of software objects in a transparent way, together with the benefits inherited from attribute grammars such as functional description and locality of description. Moreover, the abilities to describe and support changes of object oriented software processes provide a flexible way for software process evolution, which is not available in process models with clear distinction between processes and products.

As for realizing the evaluation of dynamic changes, there are also many methods proposed. For example, in objectbase management systems, eager instance conversion[PS 87], logical instance conversion[BKKK 87], version interface mechanism[SZ 86] and other methods dealing with object definition changes have been proposed and used. In [TK 89], we have argued that our meta mechanism with *lazy instance conversion* provide for clearer design concepts and more flexibility. In this paper, we have specifically explained that, by using meta-objects, the change

management and version control are masked from the object definition level, thereby leaving users a clear view of the objects. Also, meta-objects are flexible for implementing and dynamically modifying the system since they can easily be re-linked and reused.

In OS/0, we have already run several small OOAG specifications. The experiments with more practical object-base applications involving change are under way now. Those include the experiment on OOAG based object oriented software processes as described in section 4. We believe that we will find out the more concrete and practical effectiveness of our method during the experiments.

References

[Betz 85] D. Betz, XLISP: An Experimental Object Oriented Language, *Jan. 1985*

[BKKK 87] J. Banerjee, W. Kim, H.J. Kim, H.F. Korth, Semantics and Implementation of Schema Evolution in Object-Oriented Databases, *Proc. ACM/SIGMOD Annual Conference on Management of Data, pp.311-322, Feb. 1987*

[Jac 75] M.A. Jackson, Principles of Program Design, *Academic Press, London, 1975*

[Kat 89] T. Katayama, Application of Attribute Grammar Techniques to Software Development, *Proceedings of US-Japan Seminar on Software Engineering pp.91-99, 1987*

[Knuth 68] D.E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory, 2(2):127-145, 1968*

[Maes 87] P. Maes, Concepts And Experiments In Computational Reflection, *Proceedings of OOPSLA'87, Oct. 1987*

[PS 87] D.J. Penney, J. Stein, Class Modification in the GemStone Object-Oriented DBMS, *Proceedings of OOPSLA'87, pp.111-117, Oct. 1987*

[RT 89] T.W. Reps, T. Teitelbaum, The Synthesizer Generator, *Texts and Monographs in Computer Science, Springer-Verlag, 1989*

[SK 88] Y. Shinoda, T. Katayama, Attribute Grammar Based Programming and Its Environment, *Proceedings of the 21st Hawaii International Conference on System Sciences, Software Track, pp.612-620, 1988*

[SK 90a] Y. Shinoda, T. Katayama, Object Oriented Extension of Attribute Grammars and its Implementation Using Distributed Attribute Evaluation Algorithm, *to appear in Proceedings of the International Workshop on Attribute Grammars and Their Application, 1990*

[SK 90b] Y. Shinoda, T. Katayama, Software Object Modeling by an Object Oriented Extension of Attribute Grammars, *to appear in Proceedings of the Info-Japan'90, 1989*

[SZ 86] A.H. Skarra and S.B. Zdonik, The Management of Changing Types in an Object-Oriented Database, *Proceedings of OOPSLA'86, pp.483-495, Sept. 1986*

[TK 89] L. Tan, T. Katayama, Meta-Operations for Type Management in Object-Oriented Databases – – – A Lazy Mechanism for Schema Evolution, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases, pp.58-75, Oct. 1989*