代数的表現を用いたオブジェクト管理について

井上克郎　　鳥居宏次

大阪大学基礎工学部情報工学科
560 大阪府豊中市待兼山町１－１

あらまし

ソフトウェア開発に関する種々の生成物(オブジェクトという)をより形式的に取り扱うために、代数的記述言語に基づく言語ＯＡＲ（Object Management Language with Algebraic Representation）を提案する。ＯＡＲを用いてオブジェクトの生成や変更を表現式として表す方法について述べる。

Object Management Language with Algebraic Representation

Katsuro *INOUE* and Koji *TORII*

Department of Information and Computer Sciences
Faculty of Engineering Science
Osaka University
1-1 Machikaneyama, Toyonaka, Osaka 560, Japan
Telephone : 06-841-9741
E-mail address : inoue@ics.osaka-u.ac.jp

Abstract

Requirement of formal management of various software products (objects) has been increased. In this paper, we propose an object management language *Oar* (Object Management Language with Algebraic Representation), which are based on algebraic specification languages and functional programming languages. In Oar, we can easily define objects with various attributes, relations, or operations using the definitions of abstract data types. Creation and modification of the objects are performed by applying functions (operators), and the results of those applications can be seen as forms of the algebraic expressions. From these expressions, we can formally obtain and verify the characteristics of the objects.

1 Introduction

It is being widely recognized that formalizing software development processes plays the crucial roles of improving productivity and reliability of software[6]. We have studied a formalization method for the development processes with a functional programming language PDL (Process Description Language)[3]. Through this study, we knew that formalization of software products is very important as well as that of the development processes.

Software products are artifacts which are created, modified, or referred, during software developments. Specification documents, source texts, load modules, executable programs are examples of the software products. These software products are called *objects*. Object may be generally used to mean broader sense; however, our formalization for the objects can be widely applicable not only to software products but to other things which can be considered objects.

Object management is to efficiently store, retrieve, or restructure objects. Studies on the object management are initiated recently[5,9]. In this paper, we propose an object management language *Oar* (*Object management language with Algebraic Representation*), which is based on an algebraic language ASL[10] and functional language ASL/F[3]. Objects in Oar are represented as algebraic expressions, and operations to the objects are represented as function applications.

The algebraic representation of objects has benefits such that data types with operations can be abstractly defined and the semantics of the expressions are simply and formally given. In addition to these merits originated from typical algebraic specification languages, Oar has facility to dynamically create operation names, by which various constant values of a data type are easily created from one skeletal function symbol. Also it has facility to specify persistent objects to be preserved in the system. Temporary objects generated during execution could be deleted for efficiency. These facilities resolve inherent problems which would happen when we use algebraic languages for the object management.

In this paper, we also propose an architectural design of the Oar system. It bases on implementation methods for the functional languages with an environment stack, and methods for a class of algebraic languages with linked lists. The user interface of the system allows us to access objects in algebraic expressions forms.

We will discuss some requirements to the object management in Section 2. In Section 3, syntax and semantics of Oar are described. Through some examples, we will show how we manage objects in Oar in Section 4 and Section 5. In Section 6, we will show an architectural design of the Oar system.

2. Requirements of Object Management

There would be various kinds of requirements to the systems which manage various kinds of objects. For example, the data contained in a sequence of objects might be retrieved by following some predetermined structure of objects. Objects for load modules having identical attributes should be linked together. Various relations between objects such as *a_part_of*, *version_of*, etc. would be consistently maintained.

In order to satisfy these widely spread requirements, it would be insufficient for object management systems to have a single model (scheme) to handle objects. For example, a relational database model would not be powerful enough to represent hierarchy structures of objects, or a tree structure model might not allow

relations between arbitrary objects. New object management systems such as hyper media, which are quite different from existing one, would require quite different models.

To be able to use various object models in a system, it would be better to have facility to define those object models in some manners. Through the defined models, user would manage objects easily and straightforwardly according to the nature of the objects (see Figure 1).

For the purpose of defining various object models in formal ways, we would use some programming languages. Building object models would be realized naturally by using facilities to define various data types in the programming languages. Some procedural languages would be sufficiently powerful to define abstract data types, although the semantics of those procedural languages are not defined clearly and formally in general.

Algebraic specification languages have good natures, such that their semantics are defined very simply and formally, and they can define various abstract data types easily[1]. Verification can be performed very formally if needed. We have chosen an algebraic specification language as a base language to define object models, and designed object management language *Oar*.

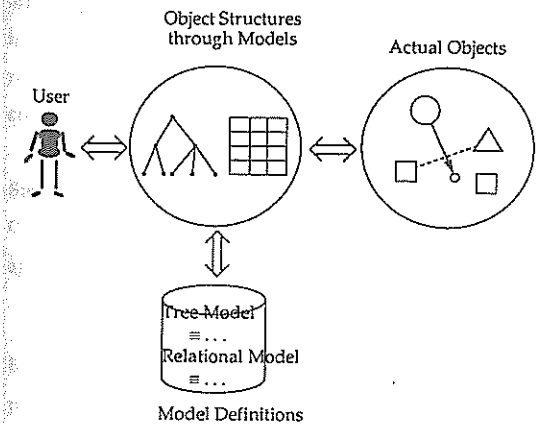

Figure 1 Objects with Various Object Models

3. Object Management Language with Algebraic Representation

3.1 Overview of Oar

In this section, we will briefly show language Oar, which has been designed for treating various software objects in formal manners.

Oar is based on algebraic specification language ASL/1 [10], functional programming language ASL/F [2], and process description language PDL [3].

The characteristics of the Oar, which are originally the characteristics of those base languages, are as follows.

(1) The semantics are defined by the equivalence relation of expressions very simply and formally. Thus, the verification of the correctness is relatively easy. The semantics of the ordinary programming languages are generally based on the behavior of compilers or particular processors, and they are more complicated.

(2) We can write descriptions in various abstraction levels. In any abstraction level, the description is meaningful in the sense that the semantics of those descriptions are formally given.

```
1    new_stack    |                    ->    stack
2    push         |element, stack      ->    stack;
3    pop          |stack               ->    stack;
4    top          |stack               ->  element;
5    [    pop(push(E, S))    ==    S     ;]
6    [    top(push(E, S))    ==    E     ;]
```

Figure 2 Example of Stack in Oar

3.2 Syntax and Semantics

Figure 2 shows a simple example of an Oar program for the definition of stack. From line 1 to 4, data types of the domains and ranges are

declared for each function. Line 5 and 6 are axioms defining each function using function symbols and variables.

We will describe two major components of Oar, type declarations and axioms.

Type Declaration

For each function, there is one type declaration statement, which shows the data type of domains (arguments) and ranges (function values) in the following ways.

```
f | type1, type2, ..., typen-> typef;
```

where *type1, type2,...* are the domain types and *typef* is the range type.

These type declarations for the built-in functions in the system are shown to the users if required. Those for user defined functions are supplied by the users.

Axiom

Axioms which define the functions are denoted by

```
[ h(X1,X2,...) == expression ;]
```

or

```
[ f(g(X1,X2,...)) == expression ;]
```

Here, X_1, X_2, ... are distinct variables, h, f, and g are function names, and *expression* is one of the following; a variable X_i, a constant, a built-in function including *if then else*, a user defined function, or a composite expression of these. If no variables appear, the axiom is a definition of a *constant*.

To simplify the implementation, we limited the left-hand side forms to simple functional styles, or simple constructor styles[10]. The same function names cannot appear twice as outer most function symbols of the left-hand sides of axioms. Functions appearing as only inner function as g is called *constructor*. The constructors may appear more than one left-hand side expression.

The semantics of an Oar expression are defined by the equivalence relation created by the axioms[10]. The value of an expression is a single constant expression or an expression with a constructor as the outer most function symbol, which is equivalent to the expression. The axioms for built-in functions are assumed to exist, however, they are not used for implementation.

3.3 Characteristics of Oar

In addition to the characteristics of ordinary algebraic programming languages and functional languages, Oar has special features useful to object management.

1) With help of Oar programming system, we can create various constant objects. In the ordinary algebraic and functional language systems, we have to prepare distinct constant functions for each different object. In Oar, however, using skeletal constant function, we can create unique objects very simply as discussed in Section 4.2.

2) When we need to modify objects, no original objects would be necessary after the modification in many cases (although there may be exceptional cases such that we intentionally keeps older versions for making backup copies or other reasons).

On the other hand, when we perform these operations on the algebraic or functional languages, the original objects represented by the expressions might be still preserved. Therefore, if we evaluate Oar expressions by using ordinary implementation methods of algebraic and functional languages, many copies of objects are generated and the execution would be very inefficient.

To resolve this problem, Oar allows users to specify object names to be preserved until the execution terminates. If there is no specific indication of the preservation, those objects are deleted as temporary objects (discussed in Section 5).

4. Interactive Operation of Objects

4.1 Example of simple C program development

As we have discussed in previous sections, expressions of Oar represent objects such as files in an actual computer system. In this section, we will see how the expressions are created under development processes of a C program. At first, a specification for the objective program is made (step 1), next C source program for the specification is developed (step 2), then the C program is compiled (step 3), and finally the load module is linked and an executable program is obtained(step 4). Figure 3 shows the these operations and the obtained objects.

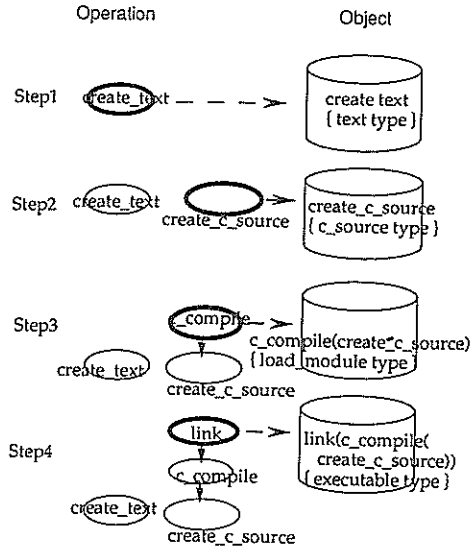

Figure 3 C Program Development

Step 1

Create a text-type object. The user looks for a suitable operation (function) which creates text type objects. The Oar system displays the possible operations whose range type is the text. Assume that the user selects *create_text* defined as follows.

```
create_text |      ->     text  ;
[     create_text == exec(editor)    ;]
```

By applying this *create_text*, a text editor is activated according to the axiom. *Exec* is a built-in function which activates a tool specified by the argument. In this case, a text editor in the system is invoked and a text created by the editor is returned. This text is now represented as *create_text*.

Step 2

Create C source program. The user selects and applies *create_c_source* defined as follows.

```
create_c_source |       ->     c_source     ;
[create_c_source== exec(editor);]
```

By this application, a text editor for C source programs is invoked. After the completion of the editing, the created object of a C source text is represented as *create_c_source*.

Step 3

Compile the C source program. The user selects an operation *c_compile* defined as follow.

```
c_compile    | c_source  ->load_module      ;
[c_compile(X) == exec(cc,   X)  ; ]
```

Here, X represents a variable, which is replaced with an actual object of the C source type. In what follows, a single capital character denotes a variable. By applying this operation with object *create_c_source* as X, C compilation is performed. A *load_module* object obtained is expressed as follows.

```
c_compile(create_c_source)
```

Step 4

Link the load module. Operation *link* is chosen and applied with the load module object.

```
link| load_module -> executable          ;
{     link(X)      ==     exec(ln, X) ;]
```

After completion of these four steps, we get four objects represented by the following expressions.

```
create_text
create_c_source
c_compile(create_c_source)
link(c_compile(create_c_source))
```

These four objects directly show the characteristics and the history of operations applied to the objects.

4.2 Unique operation names

In the previous example, we simply create one object of the specification text. We may want to create more than one object of the specification. It would be considered that we simply repeat Step 1 to get more objects; however, repeating the same steps creates more than one object having the same object name (i.e., same expressions), *create_text*, although we may intend to have distinct object names.

To prevent this object name crush, we have devised a mechanism for assigning unique object name to each object. The Oar system always provides unique operation names when the user selects an operation to apply, instead of always displaying same operation names as shown in Section 4.1. For example, the operation shown at Step 1 was

```
create_text |     ->    text  ;
```

Instead, a unique operation name actually the system provides is as follows:

```
obj1:create_text  |     ->    text  ;
[     obj1:create_text== exec(editor);]
```

Here, the unique name is *obj1:create_text*. It is composed of a variable part (a unique part) *obj1*: and a fixed part (a skeletal part) *create_text*. The unique part is generally represented by *obj1*:, *obj2*:, ... , *obji*:, ... , in this paper. It is automatically supplied by the Oar system or specified by the user. The user may choose either a supplied name or his/her favorite name as the unique part. If the user's favorite name is not unique in the system, i.e., the same unique part is already used, the system warns. The overall name is an operation which creates a unique constant of the text type.

The object created by this constant operation is also *obj1:create_text*. We can simply use an abbreviation form *obj1* for this object. When we repeat Step 1, the system provides the same skeletal operation name with a different unique part such that,

```
obj2:create_text  |     ->    text  ;
[     obj2:create_text== exec(editor) ;]
```

By applying this operation, the editor is again invoked and a different object *obj2:create_text*(or simply *obj2*) is generated.

Using this mechanism, revisions of objects are easily realized. An object already created is modified, and a new object is created. The modification of the text type object is performed by operation,

```
obj3:modify_text  | text -> text ;
[obj3:modify_text(Y) == exec(editor, Y);]
```

Here, *obj3* is a unique part given by the system, and application of this operation with *obj2* generates object *obj3:modify_text(obj2)* (or simply *obj3*).

4.3 Overloading operation names

For the cases such that an executable program consists of several modules as discussed in the previous section, first we would separately compile each module and next link them together. These operations are expressed in Oar as follows.

```
obj1:create_text  /* Specification of module 1 */
obj2:create_c_source     /* C source of module 1 */
obj3:c_compile(obj2)     /* Compile and generate
              a load module of module 1 */
obj4:create_text  /* Specification of module 2 */
obj5:create_c_source     /* C source of module 2 */
obj6:c_compile(obj5)     /* Compile and generate
              a load module of module 2 */
obj7:link(obj3, obj6)    /* Create an executable
                            object */
```

In this case, operation link has two load modules as its arguments. Since it is not practical to limit the number of arguments to two, we allow to take arbitrary number of arguments by using overload of the function names defined as follow.

```
obji:link    |     load_module -> executable;
obji:link    |     load_module, load_module ->
                 executable  ;
      ......
```

4.4 User defined types and constructors

In Section 4.5, the data types and functions on them are provided by the system. In addition to these types and functions, users can define new types and functions on them.

Consider the example of C program development. If we may want to combine together a text for specification, a C source text and a load module, into one module, we will define a new function *mk_module*, which creates a new type *module*. This *module* is a tuple of those three elements, and the type declaration would be as follows.

```
obj4:mk_module | text, c_source, load_module
                    -> module          ;
```

For example, let *obj1*, *obj2*, *obj3* be the types of *text*, *c_source*, *load_module*, respectively, and applying this function creates object *obj4:mk_module(obj1, obj2, obj3)*.

Extracting a component from a module requires additional function *get1*, *get2*, *get3*. Together with these functions, axioms for *mk_module* is defined as a constructor.

```
obj5:get1   |    module->    text   ;
obj5:get2   |    module->    c_source;
obj5:get3   |    module->    load_module ;
[    obj5:get1(mk_module(X,Y,Z)) == X    ]
[    obj5:get2(mk_module(X,Y,Z)) == Y    ]
[    obj5:get3(mk_module(X,Y,Z)) == Z    ]
```

Note that no unique names are attached to *mk_module*.

When a constructor function is apply to an object, the system does nothing but simply generates an expression, such as *obj4:mk_module(obj1, obj2, obj3)*. When an extracting function such as *get1* is applied, expression *obj5:get1(mk_module(obj1, obj2, obj3))* is first made. Since it is equivalent to *obj1* by the axiom, the system tells users that *obj5* is equivalent to *obj1*, and it changes the display name from *obj5* to *obj1*.

4.5 Built-in data structures

Commonly used data types such as list are provided by the system as system built-in data structures (types). Users can use these data structures without declarations of the data types and functions on them.

For example, a list of any objects can be created by performing built-in function *cons*. If the type of the element is, say, *load_module*, the object *cons(obj1, nil)* will be of the type of *list_of_load_module*. The users can also use *car* and *cdr* for this *list_of_load_module* without declarations. We cannot append different data types such as *c_source* to *list_of_load_module*.

5. Programming in Functional Style

We have discussed how to define data types and how to create objects interactively. In this section, we will discuss how to define functional programs which perform various operations on objects. Using functions and data types already defined as discussed in previous sections, we can define the functional programs.

5.1 Simple iteration

Consider a case such that there is a list of C source programs, and each of the elements is to be compiled into load modules and connected together into another list of the load modules. For this purpose, we define a function *compile_all* as follows.

```
obji:compile_all
     | list_of_c_source_program ->
                  list_of_load_module;
[ obji:compile_all(L) == if null(L) then nil
     else cons(c_compile(car(L)),
          compile_all(cdr(L)))]
```

After defining this function, the user applies this function to an existing list of C source programs.

The following example shows a process to repeat text editing until a condition is satisfied.

```
obji:repeat |     text          ->    text   ;
[ obji:repeat(X) == if cond(X)        then  X
          else repeat(modify_text(X))    ]
```

Note that each time when *modify_text* is

invoked, the generated object is not preserved except for the final result returned as *obji*. This happens when executing a function having the same range and domain types, and conditions for 'globalization' are satisfied[2]. This implementation method helps to execute the programs efficiently since there is no need to remember each copy of original values. However, it would cause some inconsistency for the programs such as shown below.

```
obji:f|text          ->     list_of_text;
[obji:f(X)    ==cons(modify_text(X),
              cons(modify_text(X), nil))]
```

In this case, *modify_text(X)* is evaluated twice if no optimization for eliminating duplicate computation is performed[2]. If we assume that an expression is evaluated from the left to right, the right expression *modify_text(X)* is evaluated later, and when the right expression is evaluated, object *X* has been already modified and no original object exists.

To prevent this problem, when the globalization conditions are not satisfied, we will specify them on the definitions of those functions with key word *obj:* shown as follows

```
obji:f       |text ->     list_of_text;
[obji:f(X)   ==     cons(obj:modify_text(X),
             cons(obj:modify_text(X), nil))]
```

The system keeps the original objects when it evaluates the sub-expressions with *obj:*, and the objects created for those subexpressions are named with the system-assigned unique names.

5.2 Sequence of development processes

Consider the development processes shown in Section 4.1. In that case, we first get a text of specification, a C source program, a load module, and finally an executable object. Since each operation is applied manually step by step, there was no need to have control mechanism for these application order.

If we want to perform those application automatically, we will define user defined functions, which will create these objects as the result. Consider the following definition of *g*, which creates those objects as a form of a tuple with type name *tup*.

```
g|  -> tup  ;
[obji:g() == mk_tup(
      obj:create_text,
      obj:link( obj:c_compile(
                obj:create_c_source)))  ]
      mk_tup | text, executable -> tup;
      [get1(mk_tup(X,Y))==X          ]
      [get2(mk_tup(X,Y))==Y          ]
```

Although the application of this constant function *g* generates the four objects we want, the order of invocation of two editors for the specification text and the C source program are not determined (actually, it would be determined by the system implementation ways).

If the invocation order is essential to create the objects, we consider that an object created by a later invocation depends on the previous one and the later invocation takes previous objects as its argument .

To realize this, we modify *create_c_source* and define *create_c_source'* shown below.

```
obji:create_c_source', text -> c_source;
[obji:create_c_source'(X) == exec(editor)]
```

In this definition, variable X does not affect the result. If we assume that the functions are called by value, the editor is not activated until object *X* is passed to the function. Using this function, we can redefine *g* as follows.

```
g|  -> executable ;
[obji:g() ==
      obj:link(
            obj:c_compile(
obj:create_c_source'(obj:create_text)))   ]
```

6. Implementation

In this section, we will discuss how the Ora system is implemented on actual computer system. We may think of various kinds of architectural design. Figure 4 shows one candidate architectural design.

1) The *user interface* provides a visual interface between the Ora system and the users. In order that the user is able to find out easily appropriate

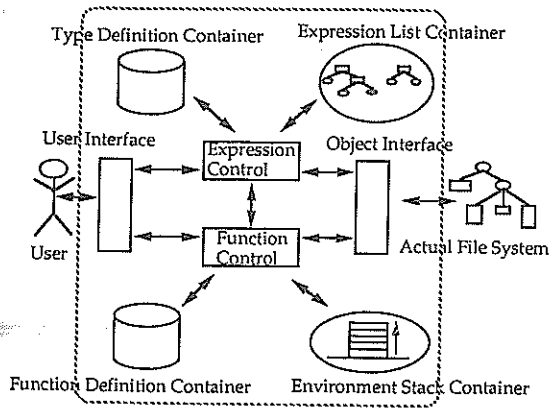

Figure 4 Architectural Design of Oar System

operations to apply, the user interface shows possible operations searched by the operation names, the data types of the ranges, or those of the domains. The users are not necessary to input algebraic expressions, but simply selects a possible operation. It also displays current objects existing in the expression list container in various ways, such as expressions sorted by the object names or tree structures showing relations among objects.

2) The *type definition container* stores definitions of the built-in data types and user defined data types (i.e., definitions of constructors and extraction functions). It keeps both type declarations and axioms.

3) The *expression list container* holds lists representing the objects which are created by the user's interactive operations or by the applications of user defined functions. A node in the list may have an object name annotated automatically by the system or by the users. More than one list could co-exist in the container at the same time, and the user can specify and remove some of them from the container.

4) The *expression control* controls the type definitions and the expression list container. It checks the equivalence relations and renames objects, when an extraction function for constructors is applied. It also generates unique object names to distinguish newly created objects from existing ones.

5) The *function definition container* contains the definitions of user defined functions which are recursive forms or simple calls to the built-in functions such as tool (utility program) invocation function *exec*. This container keeps both type declarations and axioms.

6) The *environment stack container* has a stack which stores values for current environment, such as actual parameters or temporary results of the evaluation.

7) The *function control* is responsible for the execution of user defined functions. It controls mutual and self recursive activations of the functions. It also adds, deletes, and modifies the definitions in the function definition container.

8) The *object interface* maps the expression lists into files on the actual file systems. A node in the expression list container, corresponding to a constructor, is not mapped to an actual file, since creating a file to remember information of only a constructor is expensive. Other objects such as texts of specifications and C source programs are mapped on the actual file system.

7. Conclusions

We have discussed the concepts of object management language Oar and the Oar system. By using the Oar system, we can formally define various object schemes, and can naturally manage objects based on those schemes. Formal verification of characteristics of objects can be performed relatively easily using various verification techniques of algebraic specifications.

We are currently doing detail design of the Oar system and will start implementation of a prototype. It will work stand-alone as an object management system, and also it will work with our PDL system[3] which manages software development processes. Together with the PDL system, the Oar system would be a prototype of Software Designer's Associates[5, 9, 11]. We are also developing a relatively large examples of Oar programs. This program will manage not only objects but also development processes associated with objects.

Acknowledgement We are grateful to the members of Software Designer's Associates Consortium whose discussions contributed to the design of our language and system. Especially, authors would like to thank W. E. Riddle, L. G. Williams, J. H. Sayler, and K. Kishida. Many of the ideas in this paper are results discussions with them.

References

[1] Ehrig, H. and Mahr, B., "Fundamentals of Algebraic Specification 1", Springer-Verlag, 1985.

[2] Inoue, K., Seki, H., Taniguchi, K., and Kasami, T., "Compiling and Optimizing Methods for the Functional Language ASL/F", Science of Computer Programming, 7, 11, pp.297-312, Nov. 1986.

[3] Inoue, K., Ogihara, T., Kikuno, T., and Torii, K., " A Formal Adaptation Method for Process Descriptions", Proceedings of the 11th International Conference on Software Engineering, Pittsburgh, PA, pp. 145-153, May 1989.

[4] Kim, W., Banerjee, J., Chou, H., Garza, J., and Woelk, D., "Composite Object Support in an Object-Oriented Database System", Proceedings of OOPSLA-87, Orlando, FL, pp.118-125, October 1987.

[5] Kishida, K., Katayama, T., Matsuo, M., Miyamoto, I., Ochimizu, K., Saito, N., Sayler, J. H., Torii, K., and Williams, L. G., "SDA : A Novel Approach to Software Environment Design and Construction", Proceedings of the 10th International Conference on Software Engineering, Raffes City Singapore, pp. 69-79, April 1988.

[6] Osterweil, L., "Software Process Are Software Too", Proceedings of the 9th International Conference on Software Engineering, Monterey, CA., pp.2-13, April 1987.

[7] Patel, S., Orr, R., Norris, M., and Bustard, D., "Tools to Support Formal Methods", Proceedings of the 11th International Conference on Software Engineering, Pittsburgh, PA, pp. 123-132, May 1989.

[8] Penedo, M. H., and Riddle, W. E., "Software Engineering Environment Architectures", IEEE Software, 14, 6, pp. 689-696, June 1988.

[9] Riddile, W. E., "Software Designer's Associates : A Preliminary Description", Proceedings of the 20th Hawaii International Conference on System Sciences, Kona, Hawaii, pp.371-381, January. 1987.

[10] Torii, K., Morisawa, Y., Sugiyama, Y., and Kasami, T., "Functional Programming and Logic Programming for the Telegram Analysis Problem", Proceedings of 7th International Conference on Software Engineering, Orlando, FL. pp.57-64, March 1984.

[11] Williams, L. G., "Object Management Issues for Software Designer's Associates", Software Engineering Research, Boulder, CO, July 1988.

盛光印刷所