Structuring and Derivation in Algebraic Specification/Programming Language Systems[1]

KOKICHI FUTATSUGI*

We have designed, implemented, and experimented with two language systems based on algebraic specification techniques: HISP [4, 5], and OBJ2 [3, 7].

One of the main purposes of HISP and OBJ2 is to support the so-called upstream software development process in a rigid and formalized fashion; for example, in the interactive development of formal specifications. We have experimented with these two language systems the writing of several kinds of small to medium-size software specification.

In this paper, we review our experiences with these two language systems and envision desired features of future algebraic specification and/or programming language systems. The review will be done through analyses of the following two important technical issues:

- structuring of specifications and/or programs
- interactive derivations of specifications and/or programs with structure.

1. Introduction

Algebraic specification techniques are now considered as one of the most promising approaches to specification writing in general. They were introduced around 1975 as a method for specifying so-called abstract data types [15, 18]. Substantial research efforts were invested in a wide range of areas from basic algebraic semantics theory to the application to software production processes.

HISP (Hierarchical Specification and/or Program Processor) [4, 5] was designed according to the idea that "a large part of software systems can be described as a hierarchical structure of abstract data types." The idea was embodied as a language in which each software module (description unit) is modeled as an abstract data type with hierarchical structure. In HISP, each software module is the result of applying one of several module-building operations to already existing modules. This basic feature of the language makes it possible to write inherently hierarchically structured software. Using this property, many mechanisms for top-down software development are easily realized. Parameterized types, in particular, are available in the language by using these specific operations for module building.

OBJ2 was designed and implemented in 1984 at SRI International by K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer [7] as a successor to the previous OBJ family of languages [13, 14]. OBJ2 is an algebraic specification based ultra-high-level specification and/or programming language with a number of innovative features. Besides its distinctive syntax, three of the most important features are as follows:

- Its declarative, or denotational, semantics is defined by order-sorted algebras. Order-sorted algebras are a significant extension of the usual many-sorted algebras.
- Its operational semantics is defined by the term-rewriting with associative and/or commutative matching.
- It provides parameterized modules with hierarchical structures.

Recently the latest version of OBJ, called OBJ3, was released [17, 22] by SRI International. It is basically a new implementation of OBJ2, but the term rewriting interpreter has been greatly improved by the introduction of associative and/or commutative pattern matching and parameterized modules. We are now using this latest version of OBJ, and all the OBJ examples in this paper are written in it. We therefore refer to the language not as OBJ2 but just as OBJ. Activities relating to OBJ are already spreading worldwide, particulary in the U.S.A., the U.K., France, Italy, and Japan [17]. A number of attempts have been made to apply the language to several kinds of field, including the development of formal specifications, rapid prototyping, and hardware verification [26, 8, 11].

In this paper we will concentrate on the application of algebraic specification and/or programming

*Electrotechnical Laboratory (ETL), 1-1-4 Umezono, Tsukuba Science City, Ibaraki 305, Japan.

[1]A preliminary version of this paper was presented at the IFIP WG 10.1 Workshop on Concepts and Characteristics of Declarative Systems (CCDS), held in Budapest, October 17-19, 1988.

language systems to the interactive development of specifications and/or programs, because this was the main motivation for our research on HISP and OBJ2. We discuss two technical issues here: the structuring of specifications and/or programs, and the interactive derivation of specifications and/or programs with structure. The environmental supports for these kinds of activity are also considered.

In Section 2 we discuss the structuring issue. Sections 2.1 and 2.2 review the basic design decisions of the structuring scheme for HISP and OBJ. A simple example is shown in two languages to give an intuitive understanding of the difference between the structuring methods in the two languages. Section 2.3 gives some considerations on how to combine the two methods.

Section 3 discusses the issue of interactive derivation of specifications and/or programs based on the structuring method of OBJ. Section 3.1 gives a tiny but suggestive example of program derivation with structure in OBJ. In Section 3.2 we first overview our experience in the usages of important features of OBJ and envision how we can exploit the powers of these attractive features in future. Section 3.2 gives arguments about the most desirable tools for future OBJ-like language systems, and summaries of basic requirements for these tools.

For the basic notions of the algebraic specification techniques underlying HISP and OBJ, we refer the reader to the classics of this field [1, 15].

2. Structuring

The most important issue in specification writing is how to give a good structure to specifications. Well-structured specifications have the following good properties:

- **comprehensible**: they are easy for readers to understand.
- **analyzable**: their properties, such as consistency and completeness, are easy to analyze.
- **reusable**: they are easy to reuse for similar but different problems.

There have been many proposals as to what constitutes a good structure for software systems. However, there now seems to be a consensus that "the hierarchical structure based on levels of abstraction" is the most promising [28]. The strength of algebraic specification techniques derives exactly from this, for they were originally invented to give rigid and formal specifications for hierarchically structured abstract data types.

HISP and OBJ2 have the same main purpose of giving good hierarchical structures to specifications and/or programs. But their stategies for achieving this goal are slightly different. We first review the basic design decisions of these two languages. A simple example is shown in two languages to give as intuitive understanding of the difference between the structuring methods in the two languages. After that, some considerations on how to combine the two methods are also given.

2.1 HISP in Brief

Some of the design decisions in HISP were inspired by Clear [1] and OBJ0 [13]. In particular, algebraic specification techniques based on initial algebra semantics were adopted in line with these two languages. Clear is not executable, but OBJ0 has executable operational semantics for interpreting equations as rewrite rules [19, 6]. HISP has similar operational semantics. This makes it possible to use HISP both for rather high-level formal specification writing and for low-level programming. Declarative semantics based on initial algebra combined with the interpretation of equations as rewrite rules provides one of the most promising bases for formalization for a wide range of software development processes.

One of the most important design goals of HISP is to make a language for specifying software systems hierarchically. To achieve this goal, HISP provides the following module-building operations. Clear's specification building operations (theory procedures) have the same purpose with different appearances.

- **Creation**: Create a new module by declaring sub-modules, sorts (data types), operators, and/or equations. Sub-modules can be considered as parameters. The Sub-module relation is not transitive.
- **Refinement**: Refine an already defined module by adding new sub-modules, sorts, operators, and/or equations to the origin-module. The origin-module can also considered to be a parameter. The origin-module relation is transitive.
- **Substitution**: Substitute other modules for sub-modules of a module. This realizes parameter instantiation.
- **Renaming**: Rename sorts and/or operators of a module.
- **Realization**: Realize operators of a module with different equations from those of the original module.

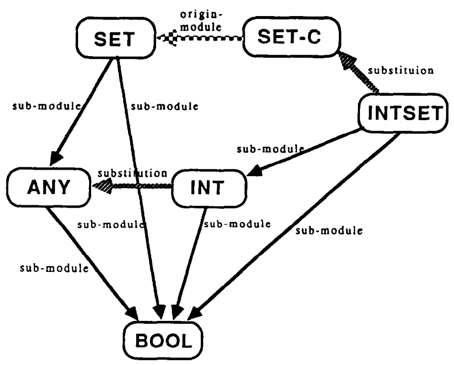

Fig. 1 INTSET in HISP.

HISP has language constructs corresponding to these module building operations.

In HISP, each software module is constructed only by using the module-building operations. As a result, the resultant system is fully modularized and parameterized. The parameterization is accomplished in two directions: the sub-module direction and the origin-module direction. Moreover, the resultant system has a hierarchical structure in these two directions.

We have written a number of examples in HISP and checked them with the symbolic executor. While most of the examples are small, there are two of moderate size. One is a semantic definition of a rather small ALGOL-like programming language, and the other is a specification of a text formatter allowing simple formula typing. We have constructed hierarchical descriptions for these examples and have seen the prospective usefulness of our method.

As a simple example, a generic, or parameterized, data type SET, its refinement, and its instantiation in HISP are shown below. /* . . . */ indicates a comment in HISP text. Figure 1 shows the overall structure of the modules in this example.

```
BOOL ::
 create                          /* create a new module        */

   sort Bool                     /* declaration of a specific  */
                                 /* kind of data called sort   */

   op   true, false: -> Bool     /* declaration of operators   */
        not _ : Bool -> Bool
        _ and _ : Bool, Bool -> Bool
        _ or _ : Bool, Bool -> Bool

   eq  var ?b?:Bool              /* declaration of equations   */
    /* as expected */
 end

ANY :: /* dummy sub-module as a parameter */
       create
         sub  BOOL               /* declaration of a sub-module */
         sort Any
         op _ =a _: Any, Any -> Bool
                                 /* predicate for identity check */
       end

SET ::
  create
    sub  BOOL, ANY
    sort Set
    op  emp: -> Set              /* empty set                  */
        _ # _ : Any, Set -> Set  /* canonical form of Set      */
                                 /* e.g. a set of 1, 2, 3 is   */
                                 /* represented as             */
                                 /*   1 # 2 # 3 # emp          */
        add: Any, Set -> Set     /* add Any to Set             */
        remove: Any, Set -> Set  /* remove Any from Set        */
        is-emp?: Set -> Bool     /* is Set empty?              */
        in: Any, Set ->Bool      /* is Any in the Set?         */
    eq  var s, t: Set; x, y: Any
    ( remove(x, emp) = emp )
    ( remove(x, y # s)
      = if x =a y then s
        else (y # remove(x,s)) fi )
    ( add(x,s) = x # remove(x,s) )
    ( is-emp?(emp) = true )
    ( is-emp?(x # s) = false )
    ( in(x, emp) = false )
    ( in(x, y # s)
      = if x =a y then true
        else in(x,s) fi )
  end
```

```
SET-C ::                        /* SET with common operator    */
  refine SET                    /* SET-C is defined by refining */
                                /* the module SET              */
    op  common: Set, Set -> Set /* the Set of common elements  */
                                /* of two Sets                 */
    eq  var s, t: Set; x: Any
    ( common(s, emp) = emp )
    ( common(s, x # t)
        = if   in(x,s) then (x # common(s,t))
                       else common(s,t)  fi  )
  end

INT :: /* INTeger */
 create
   sub  BOOL
   sort Int
   op /* as expected */
   eq /* as expected */
 end

/* parameter instantiation is realized by */
/* substitution and renaming              */
INTSET :: SET-C(* ANY <- INT; Any <- Int;
                  _ =a _ <- _ =int _ *)
            (% Set <- IntSet; common <- c %)
               /* (% ... %) is the HISP's       */
               /* renaming construct            */
               /* which can rename sorts and    */
               /* and operators                 */
```

2.2 OBJ in Brief

The main features of OBJ can be summarized as follows [7, 8, 17]:

• **OBJ** has three kinds of entity at its top level: **objects**, **theories**, and **views**.

• **Objects** declare new sorts of data, and define new operations by equations, which become executable code when interpreted as **rewrite rules**. In other words, an object encapsulates executable codes.

• OBJ's term rewriting rules use matching modulo **associativity**, **commutativity**, **identity**, and/or **idempotency**.

• **Theories** also declare sorts, operations, and equations, but these are used to define the properties of modules. That is, they define properties that may be satisfied by another object or theory.

• Objects and theories are **modules** in the usual sense. Modules can import other previously defined modules, and therefore an OBJ program is conceptually a graph of modules. Modules can be **parameterized**, and parameterized modules use theories that define both the syntax and semantics of their interfaces.

• A **module** can import other **modules** in three different ways. The three ways of importation are informally explained as follows:

protecting: import other modules as they are; the most restricted way of importation.

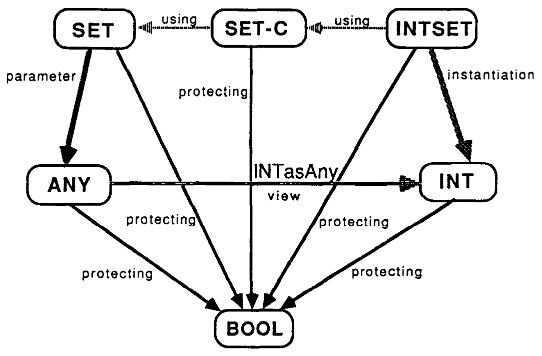

Fig. 2 INTSET in OBJ.

extending: import other modules by allowing new data items to be added to the modules, but forbidding new "meaning" to be added to already existing data items.
using: import other modules by modifying them freely.

• **Views** express bindings of actual modules to requirements, and are used to combine modules into large program units. They define the bindings of the entities declared in some theory to entities in some other module, as well as an assertion that the other module satisfies the properties declared in the theory. That is, views indicate how to instantiate a parameterized module with an actual module.

As an example of OBJ code, the code for modules that define the parameterized SET is shown. This example is intentionally made to correspond to the previous SET example in HISP. In OBJ, --- begins a comment, which ends at the end of a line. Figure 2 shows the overall structure of the modules in this example.

```
obj BOOL is       --- object BOOL is defined as follows
  sort Bool .   --- sort declaration
  op  --- as expected
  eq  --- as expected
endo

theory ANY is            --- theory ANY is defined as follows
  protecting BOOL .      --- importing the object BOOL
                         --- by protecting it
  sort Any .
  op _ =a _ : Any Any -> Bool .    --- operator declaration
endth

obj SET[X :: ANY] is
  protecting BOOL .
  sort Set .
  op emp : -> Set .
  op _#_ : Any Set -> Set .        --- canonical form of Set
  op add : Any Set -> Set .
  op remove : Any Set -> Set
  op is-emp? : Set -> Bool .
  op in : Any Set -> Bool .
  var s t : Set .                  --- variable declaration
  var x y : Any .
  eq remove(x,emp) = emp .
  eq remove(x,(y # s))
       = if x =a y then s
         else (y # remove(x,s)) fi .
  eq add(x,s) = (x # remove(x,s)) .
  eq is-emp?(emp) = true .
  eq is-emp?((x,s)) = false .
  eq in(x, emp) = false .
  eq in(x, add(y,s))
     = if x =a y then true else in(x,s) fi .
endo

--- SET with Common operator
obj SET-C [X :: ANY] is
  using SET[X] .
  op  common : Set Set -> Set .
  var s t : Set .  var x : Any .
  eq common(s, emp) = emp .
  eq common(s,(x,t))
       = if in(x,s) then (x # common(s,t))
         else common(s,t)  fi  .
endo
```

```
obj INT is  --- INTeger
  protecting BOOL .
  sort Int .
  op --- as expected
  eq --- as expected
endo

--- parameter instantiation is realized by
--- substitution and renaming
obj INTSET is
  using SET-C[INT] * (sort Set to IntSet,
                       op common to c) .
              --- * (...) is OBJ's renaming construct
endo
```

2.3 HISP Versus OBJ

As can be seen by the above parameterized SET example, OBJ can simulate a large part of HISP's module building operations by using parameterized modules. However, the basic ideas underlying the two languages are different, as summarized below.

- HISP's module-building operations are based on methodology-oriented considerations. As a result there are no serious considerations as to whether algebraic models exist for the modules obtained by using some module-building operations.
- OBJ's parameterized module is based on semantics-oriented considerations. As a result, all modules obtained by using parameterized modules have algebraic models.

Because of these differences in basic design philosophy, HISP provides a large amount of freedom in constructing modules, whereas OBJ postulates rigid rules that users must obey in order to make new modules. This implies the following differences in the usage of the two languages.

- In HISP any sub-module can be freely replaced with a syntactically similar module. This is, of course, sometimes a very dangerous operation. For example, an inconsistent pseudo-Boolean algebra with the equation (true=false) can be easily substituted for Boolean algebra, and this make the whole specification nonsense.
- In OBJ a sub-module that can be replaced should be declared with the conditions that specify what kinds of module can be substituted for the original module. Theories play important roles in stating the conditions. The replacement of a sub-module is guaranteed to be safe, if it can be validated that the substituted module satisfies the required conditions.

Generally speaking, OBJ's philosophy of providing clear and safe module-building operations should be advocated. In addition, OBJ's constructs for module structuring are powerful enough for many applications. However, there still seem to be good arguments for making it possible to write more flexible interfaces between modules. For example, we occasionally encountered a situation in which we thought it better to make a sub-module a formal parameter, without knowing how we could specify the conditions for actual parameters. Of course, once the conditions for the actual parameters are understood, it is better to write them in a rigid form similar to the OBJ style. Anyway, we think it is better for users to do more flexible structuring than is possible in the present OBJ, especially in a methodology-oriented fashion. For that purpose, the following dichotomy of concerns might be a good candidate to be adopted.

- **semantics-oriented language**: The language itself has semantically simple and clear constructs. This implies that the language does not contain powerful but dirty module-building operations.
- **methodology-oreinted environment**: Module-building or object-manipulation operations that are not necessarie explained clearly by the semantics of the language should be taken care of by the language's environment. The set of such operators takes the form of a special library constructed according to a specific methodology.

The idea of a "methodology-oriented environment" also provides a good standpoint for another but related issue of extensible/reflective language systems.

Once the meaning of some module-building operations can be understood and its semantics can be defined simply and clearly, it is better to attach constructs that support the module-building operation to the language. This kind of evolution of language is necessary in order to get usable language systems. For this purpose, the language itself should perhaps be extensible enough to allow new constructs to be attached. In other words, the language should be reflective and its entire syntax and semantics should be explainable by the language itself.

This requirement sometimes conflicts with the condition that the language itself should have simple and clear semantics. In perticular, HISP and OBJ are based on algebraic semantics, and the language constructs are basically restricted to first order. That is, the user can-not explicitly declare an operator with an operator as a argument. Higher-order functions can be handled by module-building operations or parameterized modules only in reasonably restricted manners. We believe that attaching higher-order features to the language should be done with great care. We still think that it is worthwhile to restrict the language to first-order constructs and to pursue its potential merits.

However, we also think that the language should be extensible and reflective, for the above-mentioned reasons. To resolve these conflicting requirements, it would be reasonable to make the environment or language system extensible and reflective, instead of the language itself.

3. Derivation with Structure

One of the most promising applications of algebraic specification and/or programming language systems is interactive specification and/or program derivations that make use of the hierarchical structure of modules. We are now trying to use OBJ as the basic language for formalizing so-called software development processes [27, 25] and have obtained promising results.

In this section, we will show a tiny but suggestive example of program derivation in OBJ. The example is a small program, but we think this example contains several important derivation steps that are also meaningful for more general specification and/or program developments.

Based on the example, we discuss the desirable environmental support for these kinds of derivation in OBJ-like languages.

3.1 An Example of Interactive Derivation

The problem considered is to derive, from a simple and clear definition in OBJ of sorting, a reasonably efficient sorting program, also in OBJ.

First comes the definition of sorting:

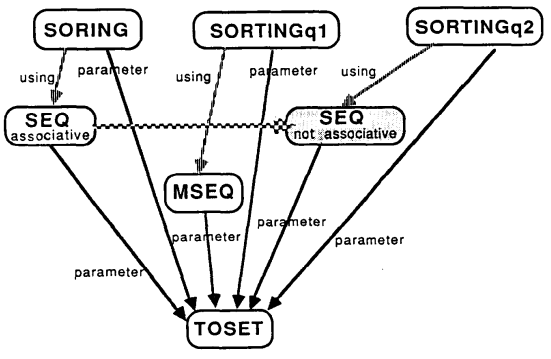

Fig. 3 Derivation of SORTING in OBJ.

```
obj SEQ[X :: TRIV] is sorts Seq .
        --- the object substituted for X
        --- must be an instance of theory TRIV
  subsorts Elt < Seq .
           --- elements are sequences
  op nil : -> Seq .
  op (_ _) : Seq Seq
             -> Seq [assoc id: nil prec 10] .
  --- concatenation operation of sequence
endo

--- total ordered set
th TOSET is sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  cq E1 < E3 = true if E1 < E2 and E2 < E3 .
  cq (E1 < E2) or (E2 < E1)
               = true if E1 =/= E2 .
                 --- if E1 is not equal to E2
endth

obj SORTING[X :: TOSET] is
        --- the object substituted for X
        --- must be an instance of theory TOSET
  extending SEQ[X] .
        --- importing the SEQ[X] by extending it
  op sorting_ : Seq -> Seq .
  op unsorted_ : Seq -> Bool .
  vars S S' S'' : Seq .
  vars E E' : Elt .
  cq sorting S = S if (unsorted S) =/= true .
        --- cq stands for Conditional Equation
  cq sorting (S E S' E' S'')
       = sorting (S E' E S' S'') if E' < E .
  cq unsorted (S E S' E' S'')
       = true if E' < E .
endo
```

From the three conditional equations in this object, it is clear that

```
if sorting(S) = T then
   1. T is permutation of S
   2. there are no subsequence E S' E' of T
      such that E' < E
```

This is the most basic definition of sorting over a total ordered set.

```
--- check that this code really works by
--- doing reduction in, say, SORTING[INT] .
--- For example, you can check that the term:
---       sorting 5 4 3 2 1 5 4 3 2 1 .
--- is reduced to (or has the simplest form of)
---       1 1 2 2 3 3 4 4 5 5
--- in the object SORTING[INT] .
```

The secret of the simplicity of the above sorting operation is that it uses the associative pattern matching of OBJ. This fact is specified by an **assoc** attribute of the operation for concatenation of a sequence. The OBJ term-rewriting interpreter can execute this sorting operation. But, as may be expected, it is inherently very inefficient.

As a first step toward a more efficient sorting operation, we fix one of the two elements compared for resolving reversely ordered pairs. This will split the sequence into two subsequences such that there is no reversely ordered pair between the two elements of the two subsequences. As a result, it is possible to call the sorting operation recursively for the two subsequences. This is a typical instance of the application of the **divide and conquer** rule. For this, we introduce marked sequence and mark the fixed element of the sequence.

```
obj MSEQ[X :: TRIV] is sorts Seq NeSeq .
  sort MarkedSeq .
  subsorts Elt < Seq < MarkedSeq .
  op nil : -> Seq .
  op @ : -> MarkedSeq .
  op (_ _) : MarkedSeq MarkedSeq ->
          MarkedSeq [assoc id: nil prec 10] .
  op (_ _) : Seq Seq -> Seq [assoc prec 10] .
endo

obj SORTINGq1[ELT :: TOSET] is
  extending MSEQ[ELT] .
  op filter_ : MarkedSeq -> MarkedSeq .
  op notYet_ : MarkedSeq -> Bool .
  var Ms : MarkedSeq .
  vars S S1 S2 S3 : Seq .
  vars E E' : Elt .
  cq filter Ms = Ms if (notYet Ms) =/= true .
  cq filter (S1 E @ S2 E' S3)
      = filter (S1 E' E @ S2 S3) if E' < E .
  cq notYet (S1 E @ S2 E' S3)
      = true if E' < E .
  op sorting_ : Seq -> Seq .
  op divide_ : MarkedSeq -> Seq .
  eq sorting nil = nil .
  eq sorting E S = divide(filter E @ S) .
  eq divide S1 E @ S2
      = (sorting S1) E (sorting S2) .
endo
```

The fact that the new sorting operation is correct with respect to the original one is checked by showing that the following equation holds by induction:

```
sorting.SORTING(S) = T
            if  sorting.SORTINGq1(S) = T
```

This fact can be tested by reducing several kinds of term, using the OBJ system. The system can also be helpful for verifying facts that can be reduced to statements that are inductive on the nested structure of terms [11].

```
--- test in a sample object,
--- e.g. SORTINGq1[INT], that the new
--- sorting is correct with respect to
--- the original sorting
--- For example, you can check that the term:
---     sorting 5 4 3 2 1 5 4 3 2 1 .
--- is also reduced to
---     1 1 2 2 3 3 4 4 5 5
--- in the object SORTINGq1[INT] .
```

We then proceed to define the filter operation without using associative pattern matching. If we can do so, we can use a normal Lisp-like list instead of a general sequence. This greatly improves the efficiency of the sorting operation.

```
obj SORTINGq2[ELT :: TOSET] is
  extending SEQ[ELT] .
  sort filterSt .
  op ___@_ : Seq Seq Elt Seq -> filterSt .
  op filter_ : filterSt -> filterSt .
  vars S S' S'' : Seq .
  vars E E' : Elt .
  eq filter (S  S' E @ nil) = (S S' E @ nil) .
  eq filter (S S' E @ (E' S'')) =
       if E' < E then filter ((E' S) S' E @ S'')
       else filter (S (E' S') E @ S'') fi .
  op sorting_ : Seq -> Seq .
  op divide_ : filterSt -> Seq .
  eq sorting nil = nil .
  eq sorting E S
       = divide(filter (nil nil E @ S)) .
  eq divide (S S' E @ S'')
       = (sorting S) E (sorting S') .
endo
```

That new filter operation is correct with respect to the original one is checked by showing that the following equation holds by induction:

```
filter.SORTINGq2(S) = T
          if  filter.SORTINGq1(S) = T
```

This fact can be tested if we reduce several kinds of term by using the OBJ system.

```
--- test in a sample object,
--- e.g. SORTINGq2[INT], that the new
--- filter is correct with respect to
--- the original filter.
```

It is clear that the sorting operation obtained is effective for an ordinary Lisp-like list. We can substitute the definitions of a sequence simply by overwriting the previous SEQ with the following SEQ. This can be done simply by feeding this code into the OBJ system:

```
--- sequence which has Lisp-like
--- last-in-first-out list structure
obj SEQ[X :: TRIV] is
  sort Seq .
  op nil : -> Seq .
  op (_ _) : Elt Seq -> Seq [prec 10] .
  op (_ _) : Seq Seq -> Seq [prec 10] .
                --- append
  var E : Elt . vars S S' : Seq .
  eq (nil S) = S .
  eq ((E S) S') = (E (S S')) .
endo

--- check that this code really works
--- faster than original one by doing
--- reduction in, say, SORTINGq2[INT]
```

We have just shown the skeleton of our interactive stepwise derivation of a sorting object in OBJ. As a matter of fact, the derivation was not straightforward, but interactions with the OBJ system make it much easier to invent, test, and verify in almost all steps of program derivations.

3.2 Toward Interactive Derivation Environment

As can be imagined from the previous example of the derivation of an efficient sorting object, the most challenging and promising application of OBJ-like languages might be interactive derivation of specifications and/or programs. Parameterized modules and subsorts are two of the most attractive features of the present OBJ language, and these features are used in very distinctive ways in the derivation processes. We first explain our experiences in the usages of these features and envision how we can exploit the powers of these attractive features. On the other hand, object management and validation tools are two of the most desirable tools for future OBJ-like language systems. We also try to summarize the basic requirements for these tools.

3.2.1 Parameterized Modules

Parameterized modules are useful in almost all phases of software development. They offer us a way of abstractly constructing such structures as list, queue, stack, and table, so that economy in the length of texts is obtained; of more significance is the fact that parameterization enhances maintainability, reusability, and encapsulation of abstract data types, since the mechanism enables conceptually identical operations, such as popping of integer stacks and of token stacks, to be made identical.

Theoretically, objects, theories, and views constitute parameterization in OBJ and contribute greatly to its expressive power. However, in our experiments, views and nonptrivial theories were seldom used as intended. Part of the reason for this is that we currently lack a usable mechanism for determining whether an object is an instance of a theory, and are therefore discouraged from defining intricate theories. Another application of parameterized objects is to allow higher-order programming within a first-order setting [10]. This application also involves the subtle problem of how to identify proper theories.

In spite of these problems, parameterized modules of OBJ are the most promising tools for formalizing specification and/or program derivation processes [8, 27]. The previous example of the derivation of a sorting program gave an intuitive feeling that parameterization would be very usable in this setting. To make this intuition more concrete, however, we need tools that allow us to define suitable theories and views.

In summary, we need to develop the following in order to allow more extensive use of parameterization:

- reasonably efficient mechanisms for checking whether an object is an instance of a theory.
- semi-automatic mechanisms to search for objects that are suitable to be substituted for a formal parameter module.
- a mechanism that is more flexible than the present theory/view mechanism for stating the interface condition between modules; this might be an augmentation of the theory/view mechanism.

3.2.2 Subsorts

Our experience teaches us that the subsort concept is very powerful and convenient, but that we must learn more about how to use it.

We tentatively divide the usage of subsorts into two classes. In one class come partial operators and error/exception handlings. It is a moot point whether we should consider all the error cases in higher-level design, which OBJ is intended to cover. A more subtle question is whether the error handlings are bunched in every object concerned. Our experiments suggest that the answer to both is yes. Errors have to be considered at some point of derivation, so we need a way to denote them. As to where to describe an error handling, the nearer to the cause of the error, the better. When we need, for instance, error message listings, they are easy to establish entirely within the normal structure of OBJ objects.

In the other class come operator inheritances. So far the use of subsorts for this purpose has been limited to cases in which the taxonomical order is clear, as in graphical entities. It may be presicely these cases that require operator inheritances, multiple or not. With this proviso, subsort declarations are very useful in establishing inheritance hierarchies with semantically clear guidelines about dos and don'ts. Furthermore, the subsort concept helps us to formalize clearly the relationships between commonly used data types and to create a taxonomy for them.

The awkwardness we encountered in describing errors in OBJ results from the inadequacy of the support mechanism for incremental object building, as discussed below. That said, subsorts provide an elegant, rigorous, and judicious mechanism for handling errors, and for making operations partial.

To develop a safer and more extensive usage of the subsort mechanism, we need to explore the following issues:

- practical knowledge on how to define errors in the usage of subsort mechanisms.

• mechanisms for supporting incremental definition of subsort relations, especially for identifying and recovering inconsistencies that may be introduced by defining a new subsort relation.

3.2.3 Object Management Tools

A supporting mechanism for describing the evolution of an object (module) and managing the evolution process will be the most important environmental mechanism in OBJ-like language systems. It is not enough that we can simulate identical results with, for example, the existing parameterization mechanism. It is crucial that almost every realistic programming derivation is incremental in some way. Among previous experiments, those describing GKS [2] and Quick Draw [26] are in this sense exceptional, in that they have had rather detailed requirement/design documentations and in that the main efforts have been to formulate the target systems in algebraic settings, not to decide what to do. Abstract data types per se, although useful in many respects, do not provide a natural way of supporting such incremental derivation processes. For this purpose, we would like to incorporate a stepwise refinement method in OBJ [25].

Considering these factors, we think that the following functions will be especially necessary for future object management tools:

• to allow users to declare correspondences (OBJ's view-like correspondences) between modules that establishe a refinement relation.

• to maintain refinement processes that allow users to undo and/or re-do the processes.

3.2.4 Validation Tools

The most important tools for any of the above-mentioned issues are validation and checking tools for objects, theories, and views. Candidates for validation/checking include the following:

• An object A is an instance of a theory B through a view C (as mentioned in Section 3.2.1).

• An object A is an refinement of an object B through a view C.

• A set of equation has noetherian and Church-Rosser properties as a set of rewrite rules.

All of the checking candidates require the implementation of some kind of induction [9, 20, 21, 24]. It is impossible to check such statements completely automatically, and we must pursue an appropriate compromise by adopting an interactive validation style.

4. Conclusion

We have reviewed the basic mechanisms of HISP and OBJ, which support structuring and derivation with the structure of specifications and/or programs. We pointed out the merits of these mechanisms, as well as problems and options for future improvements.

The motto of "putting theories together to make specifications" [1] is still the most important one in formal specification development. Clearly, OBJ, and HISP may be the only languages that provide meaningful module-building operations and have been really implemented and used (even if only in laboratories). Among these three languages, OBJ is no doubt the best as a usable specification and/or programming language. Although there is much room for improvement, as we have discussed (see also Goguen and Winkler [17]), successive versions of the new OBJ, namely, OBJ2 and OBJ3, seem to have opened up the new field of practicing by "putting theories together to make specifications" with more realistic software systems.

Acknowledgment

The contents of this paper depend heavily on the following joint work with other people.

• the design and prototype implementation of the HISP language with Mr. K. Okada and Mr. M. Ohbayashi.

• the design and prototype implementation of the OBJ2 language with Professor J. Goguen, Professor J.-P. Jouannaud, and Dr. J. Meseguer.

• the application of OBJ to several kinds of specification writings with Mr. A. Nakagawa, Mr. K. Okada, Dr. S. Tomura, and Dr. K. Ohmaki.

Without these joint projects, the analyses and discussions in this paper would not be possible.

References

1. Burstall, R. and Goguen, J. A. Putting Theories Together to Make Specifications, In Reddy, R. editor, Proc. of 5th IJCAI (1977), 1045–1058.
2. Duce, D. A. and Fielding, E. V. C. Formal Specification—A Comparison of Two Techniques, The Computer Journal, **30** (1987), 316–327.
3. Futatsugi, K. An Overview of OBJ2, Proc. of Franco-Japanese Symp. on Programming of Future Generation Computers, Tokyo, Oct. 1986, published as *Programming of Future Generation Computers*, ed. Fuchi, K. and Navat, M., North-Holland (1988), 139–160.
4. Futatusgi, K. and Okada, K. Specification Writing as Construction of Hierarchically Structured Clusters of Operators, Proc. of IFIP Congress 80, Tokyo (Oct. 1980), 287–292.
5. Futatsugi, K. and Okada, K. A Hierarchical Structuring Method for Functional Software Systems, Proc. of the 6th ICSE (1982), 393–402.
6. Futatsugi, K. and Toyama, Y. Term Rewriting Systems and Their Applications: A Survery, *J. IPS Japan*, **24** (1983), 133–146.
7. Futatsugi, K., Goguen, J. A., Jouannaud, J.-P. and Meseguer, J. Principles of OBJ2, Proc. of the 12th POPL (1985), 52–66.
8. Futatsugi, K., Goguen, J. A., Meseguer, J. and Okada, K. Parameterized Programming in OBJ2, Proc. of the 9th ICSE (1987), 51–60.
9. Goguen, J. A. How to Prove Algebraic Inductive Hypotheses without Induction: with Applications to the Correctness of Data Type Representations, Proc. of 5th Conf. on Automated Deduction, Lecture Notes in Computer Science 87, Springer-Verlag (1980), 356–373.
10. Goguen, J. A., Higher Order Functions Considered Unnecessary for Higher Order Programming, Tech. Rep. CSL-88-1, SRI International, 1988.
11. Goguen, J. A. OBJ as a Theorem Prover, with Application to Hardware Verification, Tech. Rep. CSL-88-4R2, SRI International, 1988.

12. Goguen, J. A. and Meseguer, J. Order-Sorted Algebra I: Partial and Overloaded Operators, Errors and Inheritance, Internal Report, SRI International, 1987.
13. Goguen, J. A. and Tardo, J. An Introduction to OBJ: A Language for Writing and Testing Software Specifications, In Zelkowitz, M.K., editor, Specification of Reliable Software, IEEE Press (1979), 170–189.
14. Goguen, J. A., Meseguer, J. and Plaisted, D. Programming with Parameterized Abstract Objects in OBJ, In *Theory and Practice of Software Technology*, North-Holland (1983), 163–193.
15. Goguen, J. A., Thatcher, J. W. and Wagner, W. G. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, IBM Research Report RC-6487, 1976; also in *Current Trends in Programming Methodology*, 4: *Data Structuring*, ed. Yeh, R. T., Prentice-Hall (1978), 80–149.
16. Goguen, J. A., Thatcher, J. W., Wagner, E. G. and Wright, J. B. Abstract Data Types as Initial Algebras and the Correctness of Data Representation, *Computer Graphics, Pattern Recognition and Data Structure*, IEEE (1975), 89–93.
17. Goguen, J. A. and WinklerT. Introducing OBJ3, Tech. Rep. CSL-88-9, SRI International, 1988.
18. Guttag, J. V. and Horning, J. J. The Algebraic Specification of Abstract Data Types, Acta Informatica 10 (1978), 27–52.
19. Huet, G. and Oppen, D. C. Equations and Rewrite Rules: A Survey, Tech. Rep. CSL-111, SRI International, 1980; also in *Formal Language: Perspectives and Open Problems*, ed. Book, R., Academic Press (1980), 349–405.
20. Jouannaud, J. P. and Kounalis, E. Automatic Proofs by Induction in Equational Theories Without Constructors, Proc. of Symp. on Logic in Computer Science, Cambridge, Massachusetts (June 1986), 358–366.
21. Kapur, D. and Musser, D. R. Inductive Reasoning with Incomplete Specifications, Proc. of Symp. on Logic in Computer Science, Cambridge, Massachusetts (June 1986), 367–377.
22. Kirchner, C., Kirchner, H. and Meseguer, J. Operational Semantics of OBJ3, Tech. Rep. 87-R-87, Centre de Recherche en Informatique de Nancy, 1987; extended abstract in Proc. of ICALP '88, Tampere, July 1988, to be published by Springer-Verlag.
23. Meseguer, J. and Goguen, J. A. Initiality, Induction and Computability, CSL Tech. Rep. 140, SRI International, 1983; also in *Algebraic Methods in Semantics*, Cambridge University Press (1984), 459–541.
24. Musser, D. R. On Proving Inductive Properties of Abstract Data Types, Proc. of the 7th POPL (1980), 154–162.
25. Nakagawa, A. T. and Futatsugi, K. Stepwise Refinement with Modularity: an Algebraic Approach, Proc. of the 11th ICSE (1989), 166–177.
26. Nakagawa, A. T., Futatsugi, K., Tomura, S. and Shimizu, T. Algebraic Specification of Macintosh's QuickDraw Using OBJ2, Proc. of the 10th ICSE (1988), 334–343.
27. Okada, K. and Futatsugi, K. Supporting the Formal Description Process for Communication Protocols by an Algebraic Specification Language OBJ2, Proc. of Second International Symposium on Interoperable Information System (ISIIS '88), Tokyo (1988), 127–134.
28. Torii, K., Futatsugi, K. and Mano, Y. Trends in programming methodology, *J. IPS Japan*, **20**, 1 (1979), 22–43 (in Japanese).

(Received August 31, 1989; revised May 2, 1990)