MANY-SORTED LOGICAL SPECIFICATION FOR MODULAR PROGRAMMING

Reiji Nakajima

Rapporteur: Mr. D.E. English

Abstract:

An approach to formal specification and verification for modular programming based on many-sorted first order logic is presented and discussed. The topics include: a formalism to relate specification to implementation, verification of abstract data types, type-parameterization, an interactive theorem-proving system, and formal specification integrated into a modular programming system.

1. Formal specification for modular programming

In this paper we are concerned with specification and verification for modular programming. It has been noted that the discipline of modular programming is much enhanced by providing appropriate languages (modular programming languages with abstraction mechanisms) and programming systems. With modular programming languages that have been proposed so far, a module is a syntactic unit to represent an abstraction. A module is usually divided into two components - abstract part and realization part. The abstract part defines the interface (e.g. functionality) and, if any, the specification and constitutes the visible image of the module from the other modules that invoke it. On the other hand the realization part supplies a concrete implementation of the abstraction defined by the module.

Therefore from now on in this paper by simply saying specification, we mean a specification which is presented in the abstract part of a module and specifies the abstract properties of that module. The term formal specification can mean a wide range of program documentation, from the so-called requirement specification which specifies the problem to be solved by a program, to the internal specification which more directly specifies the program behavior. The former serves as a means of communication between programmers and their customers, whereas the latter is usually used by a programmer in course of program development and maintenance.

2. Formal specification based on many-sorted logic

When a module is specified, it is expected that the specification is somehow shown to be satisfied by the implementation. Therefore, there ought to be a certain mechanically (not necessarily automatically) checkable procedure to establish this fact. If not, one can hardly discuss correctness of module implementation, nor validity of specification. Among the menu we have had, axiomatic specification methods meet this requirement. (It seems that with the

so called abstract model approach, clarity and credibility of validation proofs tends to be obscured by mingling intuitive reasonings, cf. ALPHARD). An axiomatic specification is given as a set of formulas of a certain formal system. In case it is the equational system, they are algebraic specifications, while using first-order logic, we have logical specifications. The fashion has been entirely in favour of algebraic methods these years. It appears that in some cases, by simply saying formal specification, it automatically means algebraic specification. In this paper, however, the author asserts that a very simple formalism based on the many-sorted logic can also provide a uniform, sufficient and still flexible basis for specification for moudlar programming.

Note that the choice of specification method is by no means independent of the style of programs to be specified, especially because the specification must in some way be related to its implementation, and the consistency between them ought to be established. Then program codes suitable to be specified by equational formulas will very likely look quite algebraic. On the other hand, ALGOL-like or PASCAL-like programs fit more naturally with logical specification. On the other hand, there are several applications (e.g. pointer data types [7]) for which logical methods are useful, but use of the algebraic approach seems awkward. Furthermore first order logic is theoretically well established and its understanding does not require much mathematical discipline, unlike initial algebra theories for algebraic specification.

3. Program development as a process of extending theories

As far as specification is concerned, modular and hierarchical programming using abstraction mechanism is naturally regarded as a process of extending theories in a certain formal system. The author's group at Kyoto University have designed and implemented a programming and specification language IOTA, using which one can structuredly build up theories in a many-sorted first order logic. In order to make the discussion precise we fix a deviation of the many-sorted first order logic called IL. The characteristic of IL is simply that with each sort are associated special function symbols called the primitive functions on the sort. The basic idea of data abstraction is that each data type is characterized by a set of operations and all objects of the type are generated and accessed only through these operations. The primitive functions on a sort of the logic correspond to the operations on the corresponding data type in programming. The primitive functions on a sort of IL are characterized by a set of axioms which are said to be basic on the sort.

We say the primitive functions together with the basic functions form the sort (type or sype) theory of the sort. In addition, we introduce to IL some rules of inference which are associated with some of the sorts. They are called induction rules on the sort. For instance, for the sort nn (or natural numbers) with primitive functions:

0: → nn, SUC: nn → nn, ≤ : (nn,nn) → bool, =: (nn,nn) → bool

which are characterized by the following basic axioms on nn (all free variables are universally quantified):

```
var X,Y,Z,U,V: nn
   SUC(X)=SUC(Y) ⊃ X=Y
   ~SUC(X) ≤ X
    X≤Y ⊃ SUC(X)≤ SUC(Y)
    X≤Y ∨ Y≤X
    X≤Y ∧ Y≤X ⊃ X=Y
    X≤Y ∧ Y≤Z ⊃ X≤Z
    (equality axioms)
```

There are inductions in the form of:

$$\frac{P\{0/x\},\quad P \supset P\{SUC(x)/x\}}{P}$$

which are made up of the primitive functions whose range is nn. Note here that P is any formula, x is a variable of sort nn and P t/x is the substitution of a term t of sort nn for x in P. It should be natural to assume such rules considering the underlying idea of data abstraction. Those sorts of IL with induction rules are called types whereas those without inductions are called sypes. Sype are used to provide a basis for justification of type-parameterization as we see later. In theoretical words, adoption of the inductions implies that a fixed term model domain generated from the primitive functions is associated with each type (but not necessarily with sypes).

In this paper we avoid getting into unimportant details of the syntax of the language IOTA. We simply note that a syntactic unit of the language presenting an abstraction is called a module, and that there are three kinds of modules depending on what abstraction it presents: type, sype and procedure modules. A type module presents a type theory. Sype modules will be discussed in Section 5. A procedure module presents a set of functions and their characterization axioms. (In the actual language IOTA, PASCAL-like procedures can be defined as well as functions, but in the well-known manner they are equivalently converted to functions when used in specification axioms.) These functions are not primitive on any sort nor involved in induction rules of any sort. Model-theoretically they may be regarded as standing for some functions from the term model domain of some sort to another and are somehow generated from primitive functions. The following is the theory presented by a procedure module INT SEARCH for the search function on the integers:

```
functions     SORTED: int_array → bool
              LOCATE: (int_array,int) → (bool,nn)
axioms
   var X:int_array; M,N:nn; I:int
    SORTED(X) ≡ ∀M. ∀N.(0≤M ∧ M<N ∧ N≤HIGH(X) ⊃ X[M]<X[N])
    SORTED(X) ⊃ (~LOCATE§1(X,I) ⊃ ∀M.(0≤M ∧ M≤HIGH(X) ⊃X[M]≠I))
    etc.
```

where int is the type of integers, int_array (or later in Section 5, using type-parameterization, array(int)) is the type of integer arrays. There are several built-in modules in IOTA, and programming with IOTA is to build new modules upon others starting with the built-in modules. (This is merely a formal view, though. Module building supported by the IOTA programming system is not necessarily bottom-up.) That is, as far as specification is concerned, it is to extend theories in IL starting with the combined theory determined by the built-in modules.

A module in IOTA is divided into two parts: the abstract part for presenting the theory in IL, and the realization for the implementation of the theory. Note that not every module has a realization part. If a function introduced in the abstract part of a module is only used in specification axioms and never invoked in a realization module, it need not be implemented. The function SORTED above should be an example of such. Suppose we want a type module INT_POLY for the abstract type of polynomials in a single variable with integer coefficients. For that we use three type modules NN for the type nn, INT for int and BOOL for bool. We might further want to write a procedure module INT_POLYGCD upon INT_POLY to define the gcd function on the polynomials.

Going back to INT POLY, this module introduces a new sort int_poly and its primitive functions:

```
DEG : int_poly → nn
COEF: (int_poly, nn) → int
ADD : (int_poly, int_poly) → int_poly
TERM: nn → int_poly
CM  : (int,int_poly) → int_poly
```

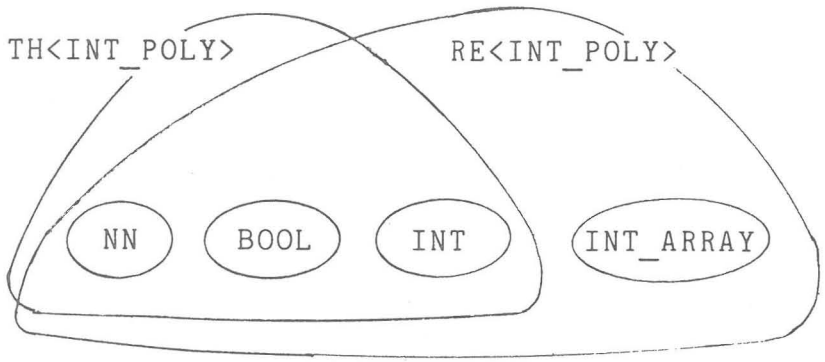
as well as the basic axioms:

```
var X,Y:int_poly; N:nn; I:int
  DEG(X)≮N ⊃ COEF(X,N)=0
  (∀N.COEF(X,N)=COEF(Y,N)) ⊃ X=Y
  COEF(ADD(X,Y),N)=COEF(X,N)+COEF(Y,N)
etc.
```

By writing INT_POLY, the combined theory determined by NN, BOOL and INT is extended to TH⊲INT_POLY⊳ by adding these functions and axioms.

On the other hand, in the realization part of INT_POLY, the abstract type int_poly is represented by another type int_array and the primitive functions on int_poly are implemented by their concrete functions defined by a body in PASCAL-like code. For instance, COEF: (int_poly, nn) → integer is implemented by its concrete function ^COEF: (int array, nn) → integer. But theoretically speaking what is a realization part? Let us regard it also as a theory. Strictly, it is not a theory of IL but of a certain many-sorted logic which

includes IL. The theory RE<INT_POLY> which is determined by the realization part of the module INT_POLY is an extension of the combined theory made of INT, BOOL, NN and INT_ARRAY. The extension is done by adding new functions ^COEF, ^ADD etc. together with their bodies as defining axioms (by way of, say, Hoare's rules or LCF). Thus we regard the implementation of an abstraction as another level of theory extension. Notice that TH<INT_POLY> and RE<INT_POLY> depend on the abstract part of the modules INT, NN, etc. but neither of them depends on their realization parts. This is the whole point of making abstraction in programs (though INT etc. are built-in in the language IOTA, so there are no such realization parts.)



4. Relating formally a specification to its implementation - verification of abstract data types

Now we want to discuss correctnesss of implementation or consistency between the abstract part and the realization part of a module, for which we must define formally the relation between specification and implementation. In order to relate the specification theory TH<INT_POLY>to its implementation theory RE<INT_POLY> we use the formalism of interpretation [6] between two formal systems. (Translating to the terminology of algebra, theory interpretation probably corresponds to theory-morphism which was independently proposed in Burstall [1] in the context of algebraic specification.)

In order to assert that a basic axiom on int_poly is satisfied by the realization part of INT_POLY, we may try to prove it in RE<INT POLY>. It is not, however, immediately possible because TH<INT_POLY> and RE<INT_POLY> are different theories. For instance, neither COEF nor int_poly in TH<INT_POLY> is in RE<INT_POLY>. Instead the concrete function ^COEF and the representing type int array are in RE INT_POLY . We must somehow translate the axioms in TH<INT_POLY> into equivalent formulas of RE<INT_POLY>.

The interpretation formalism can be applied to verification both of procedural abstraction and data abstraction, but the case of abstract data types or type modules in IOTA is particularly important.

Given a type module Q presenting the type theory of a type q, which is represented by a type rq in the realization part of Q, the interpretation mapping h: TH<Q> → RE<Q> is defined as follows:

(1) Let P, Q be formulas
h(~P) is ~h(P)
h(P ∧ Q) is h(P) ∧ h(Q)
h(P ∨ Q) is h(P) ∨ h(Q)
h(P ⊃ Q) is h(P) ⊃ h(Q)

(+) h(∀X.P) is ∀x.realize(x) ⊃ h(P{x/X}) if X is of sort q where x is a new variable of sort rq
∀X.h(P) if X is not of sort q

(++) h(∃X.P) is ∃x.realize(x) ∧ h(P){x/X}) if X is of sort q where x is a new variable of sort rq
∃X.h(P) is X is not of sort q

(2) For terms
h(F(t1,t2,...,tn)) is ^F(h(t1),...,h(tn)) if F is a primitive function of q
F(h(t1),...,h(tn)) otherwise

(3) For variables,
h(X) is X

Note that if P is a formula of TH<Q>, then h(P) is one of RE<Q>. Since we start with an axiom without free variables, (3) never occurs to the variables of sort q because such variables must have been replaced by ones of sort rs in (+) and (++) before (3) occurs. Note also that h is a finite and mechanizable transformation determined solely by what primitive functions the type q has. Here realize = realize<rq,q> is a predicate on the sort rq, as for which there are axioms and rules that are also determined automatically from the primitive functions of q. It is rather tedious to define generally, so we take as an example a very simple type module S which presents the type theory of a type s, which is represented by another type rs in the realization part. The primitive functions on s are:

F: nn → s
G: (s,nn) → s
S: s → bool

The inductions for s are:

$$\frac{P\{F(N)/X\}, \quad P \supset P\{G(X,N)/X\}}{P}$$

In order to determine what axioms and rules are necessary for realize<rs,s>, we introduce some model-theoretic discussion. (Note, however, that we never introduce the notion of domains nor the mapping between the abstract domain and the concrete domain directly

into the verification formalism. The interpretation mapping h is not a map between domains but is a translation over two formal systems. Term models are used only as a basis of justification. It is where our approach differs from the abstract model specification school.)

The term model domain of the type s is represented by the term model domain of the type rs, but actually not all objects of sort rs are involved. Since the domain for s is generated from F and G, the objects of sort rs which represent an object of s are those generated by executing ^F and ^G. The meaning of the predicate realize<rs,s> is:

for all object x of sort rs (i.e. element of the term model domain for s)

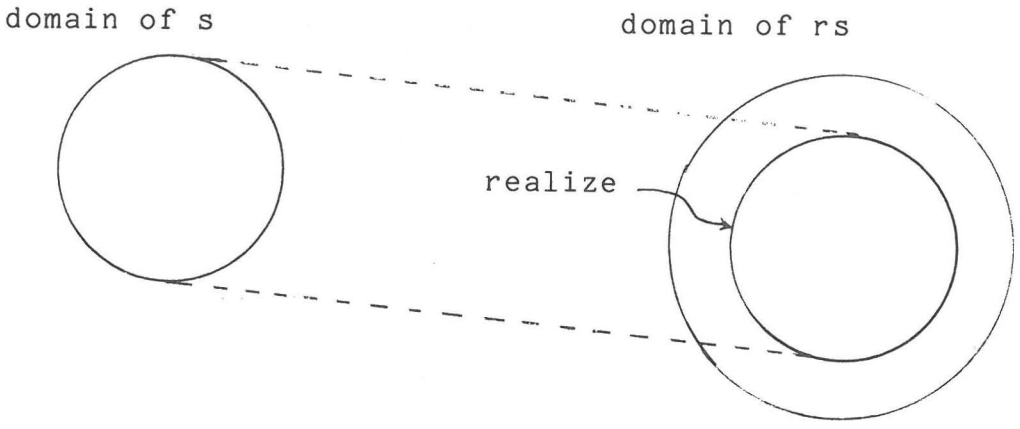
"realize<rs,s>(x) is true"
iff
"there exists an object y of sort s such that y is represented by x."

Now we have the following axioms on realize<rs,s>:

axiom var N:nn, x:rs
$\forall$ N.realize(^F(N))
$\forall$ N.$\forall$x.realize(x) $\supset$ realize(^G(x,N)),

and the following rules as a correlate of the inductions on s:

$$\frac{P\{\hat{F}(N)/x\},\quad P \supset P\{\hat{G}(x,N)/x\}}{\text{realize}(x) \supset P}$$



Using the realize predicate, a data representation invariant introduced by Hoare [2] is any predicate I(x) on the sort rs such that

realize(x) $\supset$ I(x).

Now if we add all axioms and rules of every possible realize<a,b> to the logic of IL, we are able to formulate the notion of correctness of data type representation:

The specification of a module Q is correctly implemented by the realization part of Q iff for all axioms A in the abstraction part, h(A) is provable in RE<Q> (in other words, iff the theory TH<Q> is interpreted by RE<Q>).

The procedure to establish provability of h(A) in RE<Q> is rather complicated and requires several intermediate steps. Roughly the procedure followed by the Verifier IOTA is described as follows (essentially a generalization of the proof method for data representations by Hoare [2]): First h(A) is generated from A. Then h(A) is decomposed into several formulas, each of which is proved from the programs in the realization part by way of Hoare's rules. See [4] for details.

5. Type-parameterization

Type-parameterization is an important structuring concept for formal specification. It simplifies and generalizes the specification structures and verification procedures. Also it makes libraries of specifications (or module data base in IOTA) feasible. Here we state briefly how, and with what justification, modules in IOTA can be type-parameterized.

The type theory of nn contains as a subtheory the theory of total ordering, which may be contained in other type theories. We extract such a substructure and form a sort theory. A sype order in IL has a basic theory which consists of:

primitive functions $\leq$: (order,order) $\rightarrow$ bool
=: (order,order) $\rightarrow$ bool

basic axioms

var X,Y,Z: order
$X \leq Y \wedge Y \leq X \supset X = Y$
$X \leq Y \vee Y \leq X$
$X \leq Y \wedge Y \leq Z \supset X \leq Z$
(equality axioms)

which is presented by a sype module ORDER in the language IOTA. The difference of sypes from types is that there is no induction on sypes. The relation between the sype order and the type of nn is characterized by a certain injective morphism from order to nn so that each primitive function on order is mapped to a primitive

function on nn and that each basic axiom on order is mapped to a formula that is provable in the type theory of nn. (The formal definition of the relation is given in [4].) We denote this relationship by order $\lesssim$ nn. Model-theoretically, no fixed model is associated with a sype. Rather, the models of any type q such that t $\lesssim$ q are all models for a sype t.

The same relation holds between the sype ring and the type int, too. The theory of ring is:

```
primitive functions
   +: (ring,ring) → ring, *:(ring,ring) → ring, 0: → ring, etc.

axioms
   var X,Y,Z: ring
        X+0=X
        X+Y-Y+X
        (X;Y)+Z=X+(Y+Z)
        X*Y=Y*X
        (X*Y)*Z=X*(Y*Z)
        X*(Y+Z)=X*Y+X*Z
       etc.
```

The dependence of the procedure theory of INT_POLY on the type theory of integer lies in that the latter contains the theory of ring structure as a substructure. Thus, we would rather write a type-parameterized type module POLY(T : ring) with functions

```
   COEF: (poly(T),nn) → T
   DEG : poly(T) → nn
   ADD : (poly(T),poly(T)) → poly(T)
  etc.
```

and axioms

```
var X,Y:poly(T); N:nn; I:T
   DEG(X)⩽N ⊃ COEF(X,N)=0
   (∀N.COEF(X,N)=COEF(Y,N)) ⊃ X=Y
   COEF(ADD(X,Y),N)=COEF(X,N)+COEF(Y,N)
etc.
```

where T is a formal type parameter to receive any type p such that ring $\lesssim$ p. We correspond poly(T) to a type ring_poly in IL whose theory contains such basic functions as:

```
COEF: (ring_poly,nn) → ring
ADD : (ring_poly,ring_poly) → ring
```

The induction rules on ring_poly are:

```
var X,Y:ring_poly; C:ring; N:NN
```

$$\frac{P\{0/X\},\ P\{TERM(N)/X\},\ P \wedge P\{Y/X\} \supset P\{ADD(X,Y)/X\},\ P \supset P\{C.X/X\}}{P}$$

Let us introduce a hypothetical type module RING_POLY to present the type theory of ring_poly. Now TH<RING_POLY> or the type theory of ring_poly is an extension of TH<ring>, TH<NN> and TH<BOOL> just as TH<INT_POLY> is of TH<INT>, TH<NN> and TH<BOOL>.

On the other hand any is a sype which has only

=: (any,any) → bool

as the primitive function and the equality axioms as the basic axioms. Then for any type p with equality it holds that any $\lesssim$ p. Using any, the type int_array can be generalized to a type any_array, which corresponds to a type parameterized type module

ARRAY(T : ANY)

We can continue this game to make a type-parameterized procedure module

SEARCH(T : ORDER)

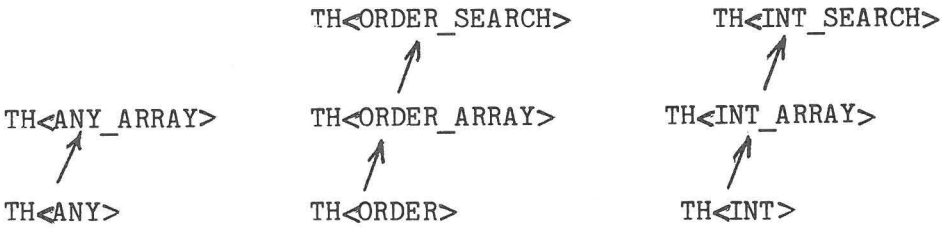
with functions:

SORTED: ARRAY(P) → BOOL
LOCATE: (ARRAY(P),P) → (BOOL,NN)

to generalize INT_SEARCH, for order $\lesssim$ int holds. This corresponds in IL to the procedure theory of ORDER_SEARCH which contains functions:

SORTED: order_array → bool
LOCATE: (order_array,order) → (bool,nn)

The type order_array (or ARRAY(T) in SEARCH(T : ORDER)) is permissible since it holds that any $\lesssim$ order (sype-sype relation).



The verification procedure discussed in the previous section can be applied to type-parameterized modules without change.

6. An interactive theorem proving system

The most crucial issue with verification for modular programming is that it requires proofs on a large number of axioms. Verification of a module can possibly involve all specification axioms of the modules on which the module is written. On the other hand, one of the main criticisms against logical specification and verification is that it requires proofs of huge formulas (which are generally referred to as verification conditions) for which there exists no decidable proof procedure. In fact, module verification with the IOTA system often ends up with a lengthy formula amounting to pages to be proven over tens of axioms without knowing whether or not it is really a theorem, although with modular programming methodology, the size of a module is supposed to be kept reasonably small.

It is absolutely true that such proofs can never be accomplished in an "out for lunch" style. With algebraic specification, however, the fact is that we are in no better position as long as there are too many axioms. In realistic program development involving a large number of modules, completely automatic proof is out of question no matter whether algebraic or logical.

A good question to ask here is why verification solely is required to be entirely automatic. Most parts of the programming activity are still left non-automated. We have at best highly interactive debugging systems but no useful automatic debugger exists yet. If verification can become realistic by the use of an interactive proof system, with the appropriate amount of human assistance, it should obtain citizenship among other programming activities.

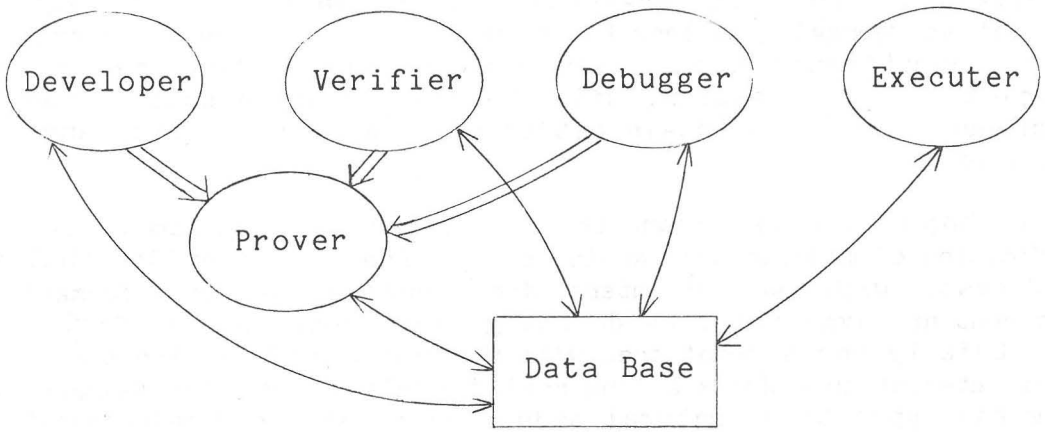
We hope to have shown that by the proof system of IOTA, verification of modular programming can be feasible provided that a proof system with powerful interactive support and proper information management are available. We do not go into details (see [3]) but state briefly how a proof goes with the Prover of IOTA. The user and system interaction reduces a long goal formula to smaller subgoals, using the opposite of natural deduction rules and domain-specific simplifiers. Ordinarily the user need not feel too much pain in this process because the system provides good interactive supports (such as a formula editor and proof-tree controller) and information management based on the module data base. Eventually, subgoals becomes small enough to prove using the automatic (resolution) prover and simplifier. To speed up the interactive process of goal-subgoal reduction, several strategies are used which utilize the fact that the axioms are hierarchically and modularly structured.

Such a proof system is rather complicated and requires some effort to learn to use. But again why must only verification systems be learned instantly, when even an editor like EMACS requires a non-negligible amount of time and mental effort to be fluent in its

use. We have observed from the experience of using IOTA in education that after a few monnths' learning, many students start being able to prove fairly large formulas with the system.

7. Iota system - integrated environment for modular program development

Programming with a modular language like IOTA can be much enhanced by providing an interactive system which is specially tailored for the language. In the development and maintenance of large scale programs involving a large number of modules, management of interrelation between modules plays crucial roles. Such global information tends to be beyond the managing ability of individual programmers. Another reason for the need of a support system is that there are different levels of concepts and activities involved with this kind of program development and maintenance. Clearly they should not all be embedded in a single programming language syntax(cf. ADA); nor should they all be taken care of by the compiler. Every concept beyond those which a programming language should include ought to be covered respectively by a meta language at the suitable level and a support system specially tailored for the programming language. A good supporting system is reallv another way to achieve "abstractions" in programming. Also uniform design of languages becomes possible by assuming that such environment and support system are to be provided.



The IOTA programming system[5] is an implemented system which supports development, maintenance, testing and verification of programs written in the language IOTA. Its functions include the management of modules and their hierarchical relations (module data base), separate compilation of (type-parameterized) modules, syntax-directed editing, and support for co-operative tasks.

The motivation for the IOTA programming system is mainly experimental and educational. We would like to see how programming methodology, language design, verification and formal specification can inpact on actual programming activities. On the other hand, we have actually used the language IOTA and the IOTA programming system in a computer programming course for senior level students at Kyoto University, in which students are expected to learn how to structure programs, write their specifications and verify them as well as learn the theoretical foundation of programming based on mathematical logic. The result has been rather successful. As already noted, many of the students become quite competent in using the language and system after a few months of learning.

Since the design of the language IOTA is mathematically oriented for the purpose of formal verification, it is not immediately applicable to many kinds of practical programming such as operating system implementation, but many design objectives and considerations adopted in the IOTA system seem to be applicable to more practical programming systems of a similar intention.

8. Formal specification integrated into the IOTA system

We have observed from actual experiences of using the IOTA system that by integrating a highly interactive modular programming system with flexible organization to support parallel development, modification and validation of programs and specifications, the usefulness of formal specifications in actual program development can be greatly increased.

The formalism for module verification discussed in Section 4 indicates that, as a formal theory, the specification is weaker than the program implementation. That is, specification is partial, or only part of the property of the program realization is determined by the specification. In particular, the specification can be empty. In fact, for specifications given by first order axioms, completeness is not a decidable property. However it seems the completeness properties are not necessarily very beneficial, as we discuss in the following.

In hierarchical program construction, many modules in the lower levels of the hierarchy are often tailored with respect to the design considerations of the upper modules. The specification of the lower modules is more or less used only for the purpose of testing and verification of the modules that depend on it. This means that the specification is expected to determine as many properties of the module as are needed to test and verify the upper modules and can be

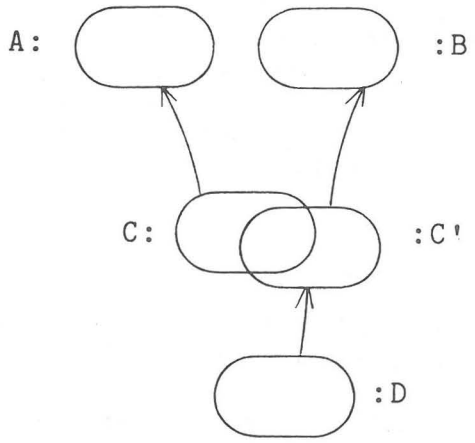
altered for this purpose. Thus the raison d'etre of the specification here is different from the usual one.

In the IOTA system the specification is regarded more as a tool for program development than as permanent documentation. The specification gives a declarative description for the modules whereas the implementation gives a procedural description. They are complementary to each other, and parallel development and modification of them, together with verification, should make it easier to guarantee correctness.

Note that the verification procedure not only validates the program realization with respect to its specification but also establishes consistency of the specification. Using logical axioms for specification, it is obviously important to ensure consistency between the axioms.

Since the specification of a module in IOTA is partial, there can be different versions of specification for a single implementation and this flexibility enhances the usage of specifications as tools as opposed to complete specifications. Note also that it is much easier and often more practical to write a partial specification than a complete one. The Developer supports such use of specifications in program development and modification.

On the other hand, the use of specification in the IOTA system can be made even more convenient by the mechanism of virtual modules. Modules once completed are subject to change at any period of the whole program development. In particular, modules in the lower levels of the hierarchy tend to be modified by the design conveniences of upper level modules even after they are completed and stored in the module data base. Instead of modifying the module itself, it is often more suitable to generate a virtual module or a variant of the original module as we state in the following. For example in the figure, the module A is completed and verified by using the specification of the module C. Now the module B is going to be written on C but B's requirement with respect to C is slightly different from that of A. To modify C in accordance with the requirement from B might destroy the work so far done on A and C. Moreover this would enlarge the size of C by increasing the number of operations and specification axioms and spoil the advantages of modular program design.



Thus it is wiser to create a slightly modified version of C to meet the requirement from B. Note that here partialness of specification provides a good deal of flexibility. Different requirements may be imposed upon a module depending on the hierarchy it belongs to. Instead of continually changing the module itself to meet the new requirements from the upper modules in different hierarchies, the user may generate variants of the original module and store them virtually. What we mean by 'virtually' here is that each variant need not actually be stored, but the system only needs to generate it when necessary. When it is determined that an object hierarchy of modules is to be processed collectively, each virtual module in it is turned into actual modules.

The merit of having virtual modules is not merely in the saving of storage. It is advantageous in that having groups of variant modules facilitates maintenance of modules. The Developer knows through what procedure a module has been changed into each of its variants, what are the differences, which parts are still valid for the variant, what remains to be done with the variant, etc. In particular, the Developer keeps information on the inter-module relations of each variant. This is useful to localize the effects of modifications. For instance in the figure, C' is a variant of C. C' may depend on a module D of which C is independent. The system can infer from the Data Base information that any change to D does not affect C (or A). On the other hand, some portion of C' can remain identical to the corresponding portion of C by an explicit declaration. This mechanism can often be conveniently used to avoid duplicating improvement efforts on that portion, which would be needed if copying was done.

For specification and verification, the virtual module mechanism makes it convenient to reuse theories and proofs with minor modification. The system attempts to see that the user need not duplicate effort to form theories and repeat proofs as long as the change does not affect the result. Also note that virtual modules provide a simple and clear-cut means to define multiple representations and implementations for the same specification or to define different specifications over a single implementation.

As is generally known, it is often the case that the major software effort is made in modifications of products in accordance with continually changing requirements rather than their initial development, and that a minor modification to a part of a large complex can cause quite unexpected global side effects. The idea behind the management of virtual modules seems to be useful in this connection.

References

[1] Burstall, R. et al: Putting theories together to make specifications. Int. Joint Conf. Artificial Intelligence, 1977.

[2] Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica 1, 1972.

[3] Honda, M. et al: Interactive theorem proving on hierarchically and modularly structured sets of very many axioms. Int. Joint Conf. Artificial Intelligence, 1979.

[4] Nakajima, R. et al: Hierarchical program specification and verification - a many-sorted logical approach. Acta Informatica 14, 1980.

[5] Nakajima, R. et al: The IOTA programming system - a support system for hierarchical and modular programming. Proc. of IFIP Congress, 1980.

[6] Shoenfield, J.: Mathematical logic. Addison-Wesley 1969.

[7] Yuasa, T.: Structural approach to pointer data types. J. of Information Processing 5, 1982

(A comprehensive description of the IOTA system is being prepared to appear as a volume in the Lecture Note series, Springer-Verlag.)