Proposal of Programming and Verification Scheme
— Program Verification Integrated
with Structured Programming —

Reiji Nakajima,† Kyoto University & Oslo University
Michio Honda,† Kagawa University & Kyoto University
Hayao Nakahara,† Kyoto University

Summary

This paper describes a programming language and a verification system to construct and prove programs with user-defined abstract data types. The design objective of the language is to uniformly describe programs, their formal specifications and supporting formal theories together with the characterization of the interrelations among these programming and verification concepts. On this language, rigorous program proofs become possible which match the modular and hierarchical program structures and concepts in the data abstraction environment.

§ 1. Introduction

The †-iota system is an integrated system for developing and verifying well-structured programs whose design and implementation are in progress on DEC 20 System at Kyoto. The † system provides an environment in which a programmer cooperates with the system to build a good program, to prove it correct and to run it. These three functional facets are undertaken by the following subsystems:

1. Program Developer [automates major part of coding job and helps the user to build structured and correct programs in intelligent ways.]

2. Verifier

3. Translator

These subsystems are highly integrated with each other on a newly developed language † to constitute the whole † system to provide an environment in which program verification is organically combined with programming methodology. This paper presents the method adopted for the † verifier.

Author's present address: † Mathematics Institute, Oslo University, Oslo, Norway, † Research Institute for Mathematical Sciences, Kyoto University, Kyoto, Japan.
† Some examples of programs and program proofs are given in the appendices.
The verification system attempts to integrate program verification with the recently noted programming methodology-data abstraction. Data abstraction was originally introduced in [1], and has been remarkably developed and established the last few years. [e.g., 7, 12]

"Data abstraction should be useful in program proving." This is the point which is generally accepted. Until now, however, this seemingly obvious assertion has been repeatedly made without any sound evidence. [In [12] efforts have been made in order to combine verification with data abstraction, but there, consideration is taken only to prove the specifications of abstract data types of restricted kind (such as queues or stacks) to be correct for their implementations using some formalized models of particular kind (such as the abstract sequence.) Nothing is said about how these specifications can be applied to prove a program of an upper level which uses the abstract types.]

No example has been given in which a complete program with user-defined abstract data types is proved correct on a logically solid foundation. No discussion has been made as to how and on which foundation the correctness of such programs should be established.

The programming and verification scheme in the verifier offers an answer to this issue. By the language and the proof method in this system, a complete program with data abstraction can be verified rigorously on a logically sound foundation.

1: With the scheme, one is able to organize various programming and verification concepts (such as operational and data abstractions, formal specifications and supporting formal theories) on all abstraction levels in such a way that the relations between these concepts and levels are clearly understood.

2: So called background theories or formal theories, which support the formal descriptions and verification discussions, are now disposed within the same syntactic frame work as the program itself, which makes the whole discussion lucid.

3: The user enjoys flexibility in that, to describe and prove part of a program, he can introduce a formal theory appropriate for the abstraction level on which the part is written.

We give the language description in §2 and the verification method in §3. The language features and the verification method, however, are closely connected and hard to be separately described.
§2. The Language - Language for describing programs, their formal specifications and supporting formal theories.

An input to the system is called a program object which consists of one or more modules. There are four kinds of modules, procedure modules, type modules, category modules and theory modules.

Procedure modules and type modules represent operational and data abstractions, respectively. A category module defines a class of data types. A theory module defines a formal theory (in a mathematical logic sense) on which program specifications are formulated and correctness proofs are conducted [See Appendix 1 for the description of category modules.]

Each module consists of an interface part and some other parts, which are either a specification part, a pre-spec part or a realization part.

An interface part is prepared for every module and contains the declaration of the operations (ops) (which are like procedures in Pascal) and functions (fns) with their domains and ranges. Syntactically, the interface part determines the external aspect of the module, i.e., only these operations and functions are visible to outer modules.

A realization part is prepared for each of the type and procedure modules and contains the implementation of the type and operational abstraction, respectively. (We borrow many of the syntactic constructs used in the realization part of a type module from CLU [4].)

A specification part is prepared for each module and contains the formal description of the module. It consists of axioms, lemmas and rules (of inference).

A pre-spec part, which may appear in a type or procedure module, contains the formal description of the module which reflects directly the implementation in the realization part. Usually, the specification part, which contains a more abstract description, is bridged to the realization part by the pre-spec part. Only the formulas in the specification part (not the pre-spec part) can be invoked in the verification discussion concerning the correctness of the other modules that refer to the module. Thus, the specification part determines the semantic aspect of the module seen from outer modules. The realization part and the pre-spec part are hidden from outer modules. To outer modules, only the interface and specification parts are visible which are independent of the actual representation and implementation taken in the realization part. [Refer to Figure 1 & 2 to materialize the language description]
We have adopted the "axiomatic description" (according to [5]) to write the formulas in the specification and pre-spec parts. (The free variables are universally quantified.) Note that the interface part provides the so called "functionality definitions" of the "algebraic specifications" proposed in [5].

In addition to the user-defined modules, some system modules are implicitly built in by the system and can be referred from all modules without explicit declaration of refer. The modules for the primitive types are such examples [Refer to Figure 3 for the system type module for sequence (array like structure of variable length)].

Finally the reader might have noticed that this language requires many syntactic redundancies. These features are intensionally introduced to clarify the program structure and concept. Since major part of coding is automatated in the system, this does not cause a burden on the user's side.

Moreover, a program object need not be complete when it is inputed to the language translator or the verifier of . Some parts of some modules may be left open. The system requires only those components which are logically needed to perform the task indicated by the user.

§3. Verification method

A. A specification part consists of some formulas which are either an axiom, a lemma or a rule (of inference).

In the case of a theory module, the axioms are implicit or explicit definitions of the functions which are declared in the interface of the module. The validity of the axioms and rules of a theory module is assumed to have been established in advance and need not be proved. In a procedure or type module, however, the axioms and rules must be proved to be satisfied by the realization part of the module. In a category module, the axioms and rules must be validated for those type modules which are declared to belong to this category.

* The rules are generated from the axioms by the "rule generator" which is built in the system.
The lemmas are formulas which are proved in the formal theory determined by the axioms and rules. When it comes to proving the specification part to be valid for the realization part, the less formulas has the specification part, the less trouble. So we want to keep the number of the axioms minimum. When the specification part is invoked in attempting to verify another module, it will be nicer if the specification part contains some more useful formulas. This is why we introduce the lemmas.

In addition to the user-defined rules, generator induction is implicitly built in each type module, which can be applied in the same way as user-defined rules.

B. Here, we describe how verification proceeds typically in °. Suppose the user wants to verify the specification part of his 'main' procedure module. The axioms placed in the specification part are to be proved correct for the realization part. For each the axioms, the corresponding verification condition (V.C.) is to be generated from the code in the realization part together with the loop invariants attached by the user.

To generate and prove the V.C., we adopt the 'top-down method' which is suggested in [3]. (Each operation call is replaced by the equivalent simultaneous execution of assignments by introducing some new functions.) Since some functions and operations defined in outer modules are called in the realization part concerned, the resulting V.C. contains these functions (some have been introduced to replace an operation). On the other hand, the axiom being proved may contain some functions defined in a theory module, and if so, these functions will appear in the V.C., too.

Now, to prove the V.C., invoked are the specification part of each module which defines some of these functions and operations. Often, to complete the proof, the user-system interaction finds it necessary to add some formulas as lemmas or axioms in some of the specification parts (each of which, of course, must be validated in due time.)

We suggest this top-down process of generating and verifying V.C.'s suits the data abstraction and modular programming environments since it reflects the modular and hierarchical program structures. [See Appendix 2 for an illustration of this process.]

On the other hand, it often happens that the proof of a program requires to be conducted on a formal theory appropriate for the abstraction level on which the program is written. Thus we introduce the syntactic concept - theory module so that the user can prepare a formal theory as appropriate.
In the conventional verification schemes, the dispositions of the supporting formal theories or background theories among the other programming and verification concepts were left rather vague. Here, after putting and arranging all thing together in the same sphere using modules and parts, we feel that the programming and verification concepts such as data and operational abstractions, their formal descriptions, and formal theories are disposed in the right position suitable for their roles. (Moreover what is striking is that, in the scheme of the verifier, there is no longer distinction between formal theories and program. They are treated in the same way.)

C. Many of the formulas, which are needed to enter into the specification part of a type module, tend to be in the form of an equivalent relation between compositions of functions (e.g. axiom 2 in specification type POLYNOMIAL in Figure 2). If one uses Hoare's system [1], however, it is generally difficult to prove such an equivalence relation to be valid for the implementations of the functions especially when the implementations contain some loops. This is the main reason why we introduce the pre-spec parts.

The pre-spec part of a type module contains such formulas that are written assuming the knowledge of the actual representation of the abstract data in the realization part, and so, is easier to prove valid for the realization part. In stead of trying to verify the axioms in the specification part directly from the realization part, one deduces them from those formulas in the pre-spec part and then validates these formulas for the realization part. This usually makes the things easier. [Appendix 4 illustrates how this verification process works.]
References


APPENDICES

Figure 1 — Program Object to Compute G.C.D. of Two Polynomials (I)

| A | B |

reads part A refers module B (represented by the interface part of B). If the interface part of a module C refers module D, it means that all parts of C refer to D. Note that a module (the specification and interface parts of it) is visible only to those outer modules (or parts) from which an arrow is directed to the module.

Figure 2 — Program Object to Compute G.C.D. of Two Polynomials (II)

```plaintext
interface category FIELD;
  thru fn ZERO: + @ as 0;
  ...
  MULT: (@, @) + @ as @@;
end interface

@ denotes one of the types which belong to FIELD.

end interface

interface category FIELD;
  var X, Y, Z: @
  axiom 1: 0 + X = 0;
  ...
end specification

interface type RATIONAL; [@ denotes RATIONAL] is FIELD
  and thru fn ORDER: (@, @) + bool as @<;
end interface
```

```plaintext
interface type RATIONAL;
  thru fn ZERO: + @ as 0;
  ...
  MULT: (@, @) + @ as @@;
  ORDER: (@, @) + bool as @<;
der end interface
```
interface procedure GCD;
  thru fn GCD: (POLYNOMIAL(RATIONAL), POLYNOMIAL(RATIONAL))
    + POLYNOMIAL(RATIONAL);
end interface

specification procedure GCD;
  refer DIVISION;
  var X,Y: POLYNOMIAL(RATIONAL);
  axiom 1: GCD(X,Y) = DIVISION GCD(X,Y);
end specification

realization procedure GCD;
  fn GCD(X,Y:POLYNOMIAL(RATIONAL)) return (Z:POLYNOMIAL(RATIONAL))
    <= avec POLYNOMIAL do
      if DEG(X) < DEG(Y) then <X,Y> := <Y,X> end if; [simultaneous assignment]
      while Y/=0 do
        X := (TERM(COEF(DEG(Y)),Y),0)*X-(TERM(COEF(DEG(X),X),DEG(X)-DEG(Y))*Y);
        if DEG(X)<DEG(Y) then <X,Y> := <Y,X> end if
      end while
    end avec
end fn

interface theory DIVISION(T:FIELD);
  thru fn GCD: (POLYNOMIAL(T), POLYNOMIAL(T)) + POLYNOMIAL(T);
  DIV: (POLYNOMIAL(T), POLYNOMIAL(T)) + bool;
  EQUIV: (POLYNOMIAL(T), POLYNOMIAL(T)) + bool as POLYNOMIAL(T)=POLYNOMIAL(T);
end interface

specification theory DIVISION(T:FIELD);
  var w,X,Y:POLYNOMIAL(T:FIELD);
  axiom 1 : DIV(X,Y) = \exists W. X=W*Y;
  ...
end specification

interface type POLYNOMIAL(T:FIELD);
  thru fn MULT: (@,@) -> @ as @*@; [MULT(X,Y) can be abbreviated as X*Y by as]
    ZERO: + @ as 0;
    COEF: (@,int) + T;
    DEG: @ + int;
  ...
end interface

realization type POLYNOMIAL(T:FIELD);
  rep *SEQ(T);
  fn MULT(X,Y:rep) return (Z:rep) [rep is like cvt in CLU]
    ...
end fn
end realization
\[ \text{specification type POLYNOMIAL(T:FIELD); } \]
\[ \text{var X,Y,Z: } \varnothing; \ldots; \]  \[ \text{[@ stands for POLYNOMIAL(T)] } \]
\[ \text{axiom 1: } \text{X*1 = X; } \]
\[ \text{2: } \text{X*(Y*Z) = (X*Y)*Z; } \]
\[ \text{... } \]
\[ \text{lemma 1: } \text{X} \neq 0 \supset \text{COEF(X,DEG(X))} \neq 0; \]
\[ \text{... } \]
\[ \text{end specification } \]

\[ \text{pre-spec type POLYNOMIAL(T:FIELD); } \]
\[ \text{refer SUM(T); } \]
\[ \text{rep = SEQ(T); } \]
\[ \text{var X,Y: } \varnothing; \text{I: int; } \]
\[ \text{avec SUM, SEQ; } \]
\[ \text{axiom 1: } \text{CONT(X,Y,I) = SUM(I,I,X,Y); } \]
\[ \text{2: } \ast COEF (X,I) = \text{CONT(X,I); } \]
\[ \text{end pre-spec } \]

\[ \text{interface type SEQ(T:FIELD); } \]
\[ \text{thru fn CONT: } \langle \varnothing, \text{int } \rangle + T; \]
\[ \text{LENGTH: } \varnothing + \text{int; } \]
\[ \text{op SET: } \langle ? \text{int,T} \rangle; \]
\[ \text{... } \]
\[ \text{end interface } \]

\[ \text{specification type SEQ(T:FIELD); } \]
\[ \text{var X,Y: } \varnothing; \text{I: int; } \]
\[ \text{axiom 1: } \text{~(PL, CONT(X,I) = CONT(Y,I))} \land \text{LENGTH(X) = LENGTH(Y) } \supset \text{X} \neq \text{Y; } \]
\[ \text{... } \]
\[ \text{end specification } \]

\[ \text{interface theory SUM(T:FIELD); } \]
\[ \text{thru fn SUM: } \langle \text{int}, \text{int}, \text{SEQ(T), SEQ(T)} \rangle + T ; \]
\[ \text{end interface } \]

\[ \text{specification theory SUM(T:FIELD); } \]
\[ \text{refer SEQ(T); } \]
\[ \text{var X,Y: SEQ(T); I,J: int; } \]
\[ \text{axiom 1: } J \not\leq 0 \supset \text{SUM(I,J,X,Y) = 0; } \]
\[ \text{... } \]
\[ \text{rule (P) 1: goal } P(\text{SUM(I,J,X,Y)}); \]
\[ \text{[P is a formula variable] } \]
\[ \text{subgoal 1: } J \not\leq 0 \supset P(0); \]
\[ \text{2: } 0 \not\leq J \supset P(\text{SUM(I,J-1,X,Y)}) + \text{CONT(X,J)} \ast \text{CONT(Y,J-J)); } \]
\[ \text{end specification } \]

- iii -
Figure 3 — System Module for Type Sequence

interface type sequence (T: any);
    thru fn create: int + @;
    length: @ + int;
    cont: (@, int) + T;
    op assign: (@|int,T); [denoted as x[i]:=exp in the actual contexts]
end interface

specification type sequence (T: any);
    var X,Y: @; I,J: int; S:T;
    axiom 1: length(create(I)) = I;
    2: 0<i<=length(X) cont(assign(X,I,S),I) = S;
    3: 0<i<=length(X) & 0<j<=length(X) & I != J
       => cont(assign(X,I,S),J) = cont(X,J);
    4: length(X)=length(Y) & i, 0<i<=length(X) => cont(X,I) = cont(Y,I))
    \[ X=Y; \]
end specification

Appendix 1 — Category Modules

CLU [4] offers a programming mechanism called "type generator" by which a
cluster can define a class of types by receiving a type as a parameter. Here,
the implementation of the cluster can not assume any structure of the type passed
as a parameter. (e.g. cluster STACK (T: type) defines the class of all stacks
whose entries contain an element of an arbitrary type T.)

This approach does not cause any inconvenience for such data types as stacks
and queues. This is not the case, however, for the type of polynomials, for example

In figure 2, module POLYNOMIAL defines the type of the polynomials with one
variable over a coefficient field T which is passed as a parameter. Obviously
the structure of T is involved in the implementation and specification of
POLYNOMIAL, i.e. each part of module POLYNOMIAL is written assuming the structure
of T as a field.

From a different stand point, it can be said that module POLYNOMIAL is written
with no assumption other than that T has the abstract property of field.
The abstract property is that, on T, are defined such functions as ZERO, ONE,
ADD, ..., MULT which are bound by such axioms as associativity and commutativity.

Category module FIELD defines a class of the data types which have this
property. The interface part of FIELD contains all the functions such as ZERO,
ONE, ..., MULT and the specification part contains the axioms mentioned above.

Now the parameter T to type module POLYNOMIAL is declared as T: FIELD.
(Our convention includes the mechanism of type generator in CLU since there is a
category any, which is the class of all types) Type module RATIONAL in Figure 2
defines one of the data types which belong to FIELD. [See the two way in which
the interface part of RATIONAL is written.]
Appendix 2 - Proving Procedure Module GCD

We want to prove the correctness of procedure module GCD which is intended to compute the g.c.d. of two given polynomial with one variable over the field rational. [Figure 1 & 2]

On module DIVISION, the formal theory of polynomial division and g.c.d. is developed, where function GCD is defined from another (predicate) function DIV (DIV(X,Y) reads \( X \) is divisible by \( Y \)). Now, axiom 1 in specification procedure GCD:

\[
GCD(X,Y) = DIVISION \# GCD(X,Y) \text{ for POLINOMIAL } X,Y
\]

asserts that function GCD computed by module GCD is \( \equiv \) to the other GCD defined in module DIVISION. (\( X \equiv Y \) means that \( X = c \cdot Y \) for some \( c \) in field \( T \))

Now to prove axiom 1, \( Z \equiv DIVISION \# GCD(X_0,Y_0) \) is the goal formula since \( Z \) receives the value of \( GCD(X,Y) \) in module GCD. (\( X_0, Y_0 \) stand for initial values for \( X, Y \)). From this goal, the V.C.:

\[
(\text{DEG}(Y) \leq \text{DEG}(X) \land GCD(X,Y) = GCD(X_0,Y_0) \land Y = 0) \supset X \equiv GCD(X_0,Y_0) \text{ etc. are generated.}
\]

are generated. Now, these V.C.'s are to be deduced from the axioms and lemmas in the specification parts of modules DIVISION and POLYNOMIAL.

Appendix 3. Proving Equalities

Here we discuss the important issue of the equality. As explained in [4], the equality predicate \( \text{equal} : (@, @} \rightarrow \text{boolean} \) as \( @ = @} \) (for the program notations, see Figure 2) is assumed to be placed implicitly in each type module.

If one wants to prove an equality \( \text{equal}(X, Y) \) on an abstract data type, this equality is to be translated into an equality on the type structure which represents the abstract data type. [In Figure 2, to prove POLYNOMIAL\# equal(X, Y) for POLYNOMIAL X, Y, realization type POLYNOMIAL is looked at. Since type SEQ is used as the representation for POLYNOMIAL (\( \text{rep} = \text{SEQ(T)} \)), \( \text{SEQ}\#\text{equal}(X', Y') \) should be proved for \( \text{SEQ} X', Y' \) representing X, Y, respectively. Now specification type SEQ is searched and axiom 1 is found which gives a condition for \( \text{SEQ} \) equality.] The user should place some axioms for the equality in the specification part of a type module if he wants to establish the equality on the data type.

Incidentally, if one wants to prove an equality on type SEQ, which is represented by a primitive type sequence, then the system type module for sequence is referred, in which axiom 4 is the equality axiom. [See Figure 3 and Appendix 4.]