A Presentation of the Specification
And Verification Project “ABEL”*

Ole-Johan Dahl and Olaf Owe

Department of Informatics
University of Oslo
Norway

November 1984

Abstract

This is an overview of the ABEL-project which is led by Ole-Johan
Dahl at Department of Informatics, University of Oslo. The first sec-
tion describes the main goals of the project. The second section de-
scribes research activities associated with the project, emphasizing
areas that are assumed to be relevant to the VERkshop. The last
section presents the ABEL85 language through a non-trivial example.

Introduction

ABEL (Abstraction Building, Educational Language) is designed to be an
integrated system for specification and programming. It contains a language
for constructive (algebraic) specification centered on the concepts of types

*ACM SIGSOFT SOFTWARE ENGINEERING NOTES vol 10 no 4 Aug 1985 page 28
(Presented at the Verification Workshop “VERkshop III”, Watsonville, CA, Feb. 85)
Reformatted.
and functions, which can be viewed as an applicative programming language. For the purposes of specification and reasoning, a version of typed first order predicate calculus that caters for partial functions is included, along with mechanisms for non-constructive axiomatics. These language elements are “non-executable”, but are supported by software for syntactic processing as well as certain kinds of semi-mechanized reasoning. For the production of software making efficient use of limited size computers, imperative programming constructs are included in the language. All language constructs can be combined in writing imperative texts in terms of non-executable concepts, either for specification purposes or as a step in a semi-formalized program development process. They may also be combined in decorating imperative executable programs with pre- and post-conditions and other assertions for Hoare-style reasoning.

The ABEL project is several years old [Faye-Schoell et al 76, Langmyhr et al 78], and ABEL85 [Dahl et al 85] is a thoroughly revised version of a language designed in the late 1970’s. The overall goal of the project is to design a language and supporting software as well as underlying theory, that

- encourage high quality specifications and programs,
- facilitate secure reasoning about specifications and programs,
- enable long term accumulation of reusable concepts,
- are well suited for teaching purposes.

An ultimate goal is to use the ABEL language and supporting system for the main programming courses given to the informatics students at the University of Oslo, Norway. We have sought to reach the goals listed above:

- by designing suitable language constructs with emphasis on encapsulation and abstraction mechanisms. There are several such constructs: contexts, properties, types, classes, and functions (including procedures). In the language design we have drawn on older programming languages like SIMULA and Pascal. Many of the constructs dealing with algebraic specification are inspired by the Larch shared language [Guttag et al 84].
• by fairly standard supporting software like syntax oriented editor, compiler, interpreter, and library system, as well as more specialized software for semi-mechanized reasoning: algebraic simplifier, theorem prover, proof checker, and verification condition generator. Some of these modules are functioning today, but not all.

**Theoretical Contributions**

The ABEL-project has encouraged theoretical research in a wide range of areas from language design to theorem proving. Fruitful work has been done in the following areas:

**Underlying logic:** We believe that it is essential to be able to reason with partial functions and partly defined expressions. Traditional first order logic is not satisfactory, since it requires total functions. We have developed an alternative first order logic, called Weak Logic, especially designed to make program reasoning as simple as possible, in particular, reasoning about “partial implementations” and partial correctness. Secure reasoning about partial implementations is important since they are found in most programs, for instance most implementations of Integers partially implement mathematical Integers, and FINSET partially implements FinSet as shown in the last section. Weak Logic is a typed two valued logic together with a “definedness” operator which enables separate reasoning about normal termination [Owe80 and Owe84]. A similar logic but with a strong interpretation of validity is being developed for the ANNA-language [Luckham et al 84 and Owe84b]. See also [Dahl84b].

**Simplification and theorem proving:** A new promising decision algorithm has been developed [Nossum84]. Its practical usefulness will be investigated. A method for simplification of quantified formulas has been found [Owe81]. For instance, all the verification conditions generated from a bubble-sort procedure were simplified to true [Owe79]. Furthermore, useful ad-hoc techniques for simplification have been developed [Nossum82, Nossum83, Dahl84a, and Thomassen84].

**Hoare-style reasoning:** The original Hoare Logic is not (relatively) complete for languages that allow partial expressions. Several modifications
that ensure relative completeness have been developed [Kirkerud82, Owe83]. The latter system is left constructive. Its proof rules are almost identical to those of the original Hoare system. However, our system is based on Weak Logic rather than traditional first order logic [Owe83 and Owe85]. A complete set of Hoare-style proof rules for communicating sequential processes (CSP) is presented in [Sound. et al 82].

**Verification techniques:** The first implementation of a verification condition generator was made in 76 (by Owe) based on left constructive proof rules. Various improvements have been investigated since then. It has been a major concern to keep generated conditions as short and simple as possible. In particular, we have developed efficient techniques to handle the problem of adaptation and the problem of generating “effect function” theorems. (An example of an effect function theorem is found in the last section.) Pre-conditions for procedure calls and loops are generated by introducing effect functions. Effect function theorems are generated by first generating a post-condition and then generating a pre-condition which in turn can be used as an effect function theorem (only loop invariants must be supplied by the programmer) [Owe85].

**Specification theory:** Guttag-style algebraic definitions have been generalized in combining several “generator” or “observation” bases. The existence of partial functions has motivated theory for reasoning with strict and non-strict equality [Owe80, Dahl et al 85]. The importance of sequences in specification and reasoning about concurrency was demonstrated by [Dahl79]. A tool box for Hoare-style reasoning about concurrency together with non-trivial applications are found in [Dahl77]. Methods for specification and reasoning with concurrency are being investigated, see for instance [Meldal84].

**An Example:**

**Specification and Implementation of Finite Sets**

The following ABEL85 declarations show an abstract definition of the concept of finite sets and its implementation by means of search trees, together
with the specification of concepts needed to verify the implementation. We shall comment on some of the main language constructs. Reserved words are written in upper case letters. First concepts of ordering are introduced:

PROPERTY PreOrd[T] ==
  FUNC T <= T =: Boolean -- the less than or equal relation
  T < T =: Boolean -- the less than relation
AXIOMS x,y,z : T
  x <= x
  (x <= y & y <= z) => (x <= z)
DEFS
  x < y == x <= y & x /= y

PROPERTY PartOrd[T]
  INCLUDING PreOrd[T] ==
AXIOMS
  (x <= y & y <= x) => x = y

PROPERTY TotOrd[T]
  INCLUDING PartOrd[T] ==
AXIOMS
  x <= y | y <= x

A PROPERTY introduces functions by specifying their signatures in terms of Cartesian products of formal and actual types. The equality and non-equality (≠) operations are always available, even for formal types. Equality is by definition a congruence relation. Semantic information about functions is provided by AXIOMS which are first order formulas, and/or constructive definitions, such as quantifier free equations.

We have above defined a hierarchy of three ordering concepts, each property inheriting all information contained in its predecessor through the INCLUDING clause. The type Boolean and its associated operations are predefined (⇒, &, → denote implication, and, or, respectively).
TYPE FinSet[T] ==

FUNC null #: FinSet -- the empty set
add(FinSet,T) #: FinSet -- add an item
has(FinSet,T) #: Boolean -- membership test

GENBAS null,add
DEFS has(null,x) == false
has(add(s,x),y) == (x = y) | has(s,y)
OBSBAS has

LEMNAS

add(add(s,x),x) = add(s,x)
add(add(s,x),y) = add(add(s,y),x)

The body of a TYPE definition can be seen as a property in which the property name is the “type of interest”, the one being defined. The GENERator BASis definition introduces an induction principle asserting that all values of the type of interest may be finitely generated by the functions listed. For proof purposes, the following rule is introduced.

\[ \vdash P_{null} , \ P \vdash P_{add(s,x)} \]
\[ \vdash \forall s : FinSet : P \]

The semantics of the has function is given by inductive definitions using generator induction with respect to the first argument. In general, generator induction nested to any depth is allowed in definitions.

The definition of OBServation BASis defines equality over FinSet by introducing the following inference rule.

\[ \vdash \forall x : T : has(s, x) = has(ss, x) \]
\[ \vdash s = ss \]

The LEMMAS are provable by generator induction on the variable s using the given observation basis and the inductive definitions. The combination of generator basis, direct or generator inductive definitions of non-basic functions, and observation basis definition enables syntactic checks for consistency and completeness.
The concept of sequences is a generally useful one for reasoning about programs. Here we have only introduced operators which are used in the sequel. The append operator symbols are intended to indicate asymmetric plus signs. The given generator basis has the ONE-ONE property, which means that the type of interest is defined as an initial algebra. The following proof rules are introduced in addition to the induction rule.

\[ \vdash \text{empty} \neq (q \mid - x), \quad \vdash (q \mid - x) = (r \mid - y) \Rightarrow q = r \& x = y \]
Most of the non-basic functions are defined by simple generator induction. However, the singleton function could be defined directly; and nested induction was required for the nonrep-predicate. The definition of the perm-predicate is not shown. (It may be easily defined by generator induction using an auxiliary sequence function.) Two of the lemmas assert the existence of alternative generator bases. The proof system should construct the corresponding first order lemmas.

```plaintext
PROPERTY TotOrdSeq[T]  ASSUMING TotOrd[T]
  IMPORTING Seq[T] NAMED SeqT ==
  FUNC SeqT *< T  =: Boolean         -- all less than
       T <* SeqT  =: Boolean         -- less than all
       SeqT *<* SeqT  =: Boolean     -- all less than all
       inc(SeqT)  =: Boolean        -- sequence of increasing terms
       sort(SeqT)  =: SeqT           -- sort with respect to =<
... DEFs
    empty *< x == true
    (q |- x) *< y == q *< y & x < y
... Axioms
    sort(q) perm q
    nonrep(q) => inc(sort(q))
```

The element type T should be totally ordered as specified by the ASSUMING clause. For each occurrence of the property TotOrdSeq it must be checked that its actual parameter satisfies the syntactic and semantic properties of TotOrd. The type Seq[T] is made visible, and given the name SeqT, by the IMPORTING clause. The axioms of TotOrdSeq state those properties of the sort-function that are actually needed in the sequel.

```plaintext
TYPE BinTree[T]
  IMPORTING Seq[T] NAMED SeqT, Seq[{0,1}] NAMED Seq01 ==
  FUNC
    nil  =: BinTree            -- the empty tree
```
tree(BinTree,T,BinTree) =: BinTree -- construct a non-empty tree
infix(BinTree) =: SeqT -- infix traversing sequence
sub(BinTree,Seq01) =: BinTree -- pick a sub-tree
lSeq,rSeq(BinTree,Seq01) =: SeqT -- find left and right parts
-- of the infix sequence

ONE-ONE GENBAS nil,tree
DEFS
infix(nil) == empty
infix(tree(l,x,r)) == infix(l) |- x -| infix(r)
sub(nil,q) == nil
sub(t,empty) == t
sub(tree(l,x,r),d -| q) == IF d = 0 THEN sub(l,q) ELSE sub(r,q)
lSeq(nil,q) == empty
lSeq(t,empty) == empty
lSeq(tree(l,x,r),d -| q) == IF d = 0 THEN lSeq(l,q)
ELSE infix(l) |- x -| lSeq(r,q)

LEMMAS
1Seq(t,q) |-| infix(sub(t,q)) |-| rSeq(t,q) = infix(t)
sub(t,q) = tree(l,x,r) => sub(t,q |- d) = IF d=0 THEN 1 ELSE r
sub(t,q) = tree(l,x,r) => 1Seq(t,q |- d) =
      IF d=0 THEN 1Seq(t,q) ELSE 1Seq(t,q)|-| infix(l)|- x

The importing-clause introduces two versions of the Seq type, and thereby two
versions of each function defined therein. There is an identifier replacement mech-
anism in the language, which may be used to avoid name conflicts in such cases,
but in this case the following general rule of overloading resolves all conflicts: The
identity of an applied function is determined by the dominant actual parameter.
The dominant argument position is the first occurrence of the type of interest (if
any) in the signature.

The functions sub and lSeq are defined by nested induction. Since we have not
adhered rigidly to the generator induction schema, a possibility for inconsistency is
introduced. However, consistency is proved by putting q=empty in the first axiom
and t=nil in the second one of each function. Notice that the induction on q is with
respect to one of the alternative generator bases of the Seq type. The semantics
of the lSeq and rSeq functions are most easily comprehended by considering the
first lemma.
CLASS FINSET[T](N : Nat1) ASSUMING TotOrd[T] "INCLUDING TotOrdSeq[T]" "IMPORTING BinTree[T] NAMED BTree" ==

REPR m :{0..N}
   A : ARRAY {1..m} OF T
   L,R : ARRAY {1..m} OF {0..N}
"OBS Tree(k :{0..m}) == BTree ==
   IF k = 0 THEN nil ELSE tree(Tree(L[k]),A[k],Tree(R[k]))" INVAR nonrep((1 -| L[1..m] |-| R[1..m]) \ 0)
   & ALL k:{1..m}:(L[k]=<m & R[k]=<m)
   & nonrep(A[1..m]) & infix(Tree(1)) = sort(A[1..m])
"OBS has(x :T) == Boolean ==
   PROG
   IF m = 0 THEN RETURN false
   ELSE VAR i :{1..N} = 1; "q : Seq01 = empty"
   LOOP
      ASSERT ( sub(Tree(1),q) = Tree(i) &
      lSeq(Tree(1),q) *< x <* rSeq(Tree(1),q) )
      IF x = A[i] THEN RETURN true
      ELSIF x < A[i] THEN
         IF L[i] = 0 THEN RETURN false
         ELSE i := L[i]; "q := q |- 0 " FI
      ELSE
         IF R[i] = 0 THEN RETURN false
         ELSE i := R[i]; "q := q |- 1" FI
      FI
   ENDLOOP
   FI
ENDPROG
CONSTR null == PROG m :=0 ENDPROG
OPER add(x :T) == PROG ... similar to has(x), but puts x in the
   appropriate place if not already there ... ENDPROG
LEMMAS has(x) == EX k :{1..m}: x = A[k]
... FINSET PARTIALLY IMPLEMENTS FinSet

The CLASS construct is the “imperative” counterpart of TYPE. In an executable program its values can only occur as the contents of declared program
variables to be updated incrementally, they are illegal as function values. This is the recommended way of dealing with high volume data objects (in the absence of pointers).

A REPResentation is the executable counterpart of an observation basis. The INVARiant states a restriction on values attainable by the representation. In order to express the invariant, an auxiliary function Tree is needed, mapping the representation onto BTree values (abstract binary trees). This function is not intended to be part of the executable code; therefore the definition is put in quotes. The first two conjuncts of the invariant restrict the representation so that it actually does represent a tree and the Tree-function is well defined. The last conjunct expresses the search tree property.

The body of the has function is expressed in a fairly standard way, except that a “mythical” program variable \( q \) is declared and manipulated outside the executable code. It represents the access path to the current subtree and makes it possible to express the loop invariant succinctly.

The first lemma states the effect function of the has-procedure. (There would be similar lemmas for the other procedures.) The last lemma asserts the intended relationship between the class FINSET and the type FinSet. The word PARtially refers to the fact that only a subset of the abstract value set is actually implemented. (The add-procedure can at most be partially correct.)

References


[Owe79] O. Owe: “Veiledning i PROVER”, Lecture Notes, Dept. of Informatics, Univ. of Oslo, 1979 (in Norwegian)

