IST. B.BLIOTECA
Posiz. ARCHIVIO
B4 - 28

The Draft Formal Definition of Ada ®

Modeling Input-Output

DATE : *Pisa, January , 1985*

AUTHOR : *Claus Bendix Nielsen, DDC*
Nicola Botta, DDC
Alessandro Fantechi, I.E.I. - C.N.R.
Franco Mazzanti, CRAI

REPORT No :

WORKPACKAGE : *E (Trial Definition)*

DISTRIBUTION : *Internal Use Only*

This work is partially funded by the Commission of the European Communities under the Multi-Annual Programme in the field of Data-Processing, proj. No. 782. "The Draft Formal Definition of ANSI/MIL-STD 1815A Ada".

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

Mod I/O

TABLE OF CONTENTS

1 INTRODUCTION

In the Formal Definition of the Ada Language, we are interested in giving the semantics to a syntactic object (called user-program) built following an Ada grammar, but for which some semantic restrictions hold (e.g. it should be legal for every implementation , etc.)[Dyn Sem Extent].

The predefined I/O packages should not be seen as a part of the above mentioned Ada user-program (indeed it is not a user problem to define them), as they constitute a part of the predefined language environment.

Hence the behaviour modelling the activity of the elaboration of the I/O packages cannot be defined by the application of some denotational clauses to some syntactic object, but should be explicitly defined as a constant piece of the Formal Definition. In the Trial Definition a program can only make use of the generic DIRECT_IO package, whose elaboration is indeed defined by the auxiliary clause:

elab_Direct_IO: LOCAL_INF → local-inf-BEHAVIOUR

which returns the behaviour modelling the activity of elaboration of the generic package DIRECT_IO; this activity consists of the definition of the appropriate denotation corresponding to the generic package, and of its inclusion in the global environment.

Apart from the clerical work of assembling the behaviours modelling the activities of elaboration of the single declarative items into a generic package denotation, the main issues to be explored in detail are related to the definition of the behaviours modelling the activities of elaboration of the single declarative items (i.e. the type FILE_TYPE and the I/O operations), dealing with the concepts of external file and file object, with the possible concurrency of the input/output primitives and with the representation of the implementation dependent aspects .

These topics are discussed separately in the following sections.

2 EXTERNAL FILES

About the concept of file, central to the Ada I/O definition, the Reference Manual clearly distinguishes between the file (object) - that is, an object of type FILE_TYPE - and the external file - that is, an entity which is external to the scope of the language and which represents the physical file on which data are actually written or read.

The problem is whether, and to which extent, the external file concept has to be modeled in the formal definition of Ada, since it is an entity whose semantics is completely outside the scope of the language, and implementation-dependent, but on which some operations are vaguely defined in the manual.

Now, given such vagueness in the definition of external files, and the consideration that, as their name implies, external files are considered conceptually external to the program in the manual itself, we can limit our formal definition to the concepts internal to the scope of the language, modeling the operations on external files just by means of actions of the final concurrent system which represent explicit interactions with the external environment.

This choice of modeling allows also to preserve the possibility of inserting the model of an Ada program into a model of the "external environment", constituted by a larger SMoLCS system in which the program activity is composed in parallel with other activities (possibly Ada programs) of the overall system.

3 FILE OBJECTS

From the specification of the operations that can be performed on file objects, as are informally described in sections 14.2.1 and 14.2.4, it turns out that the abstract properties of type FILE_TYPE are completely defined: these properties refer to the status of the Ada file (whether open or closed, and the value of the index) and to the identification of the external file (by its NAME, etc.).

Moreover, the fact that in the manual FILE_TYPE is defined, in the private part of the I/O packages, as "implementation dependent", means only that an implementation can choose any structure to record the information needed by the allowable operation - and the minimal information needed is deducible from the manual and is not implementation dependent - plus any other information that is convenient to record for the particular implementation.

Hence the FILE_TYPE type can be given a full formal definition in terms of an abstract data type.

4 ATOMIC ACTIONS

Carefully reading the manual, we can see that nothing at all is said about the possibility that two I/O operations are performed concurrently by more than one task. Apart from the impact that such possibility could have on the external environment, which as already said is outside the scope of the language, we are concerned here with the impact on the modeling of the file object. For this purpose, we can choose among two different extreme interpretations of the manual:

a) we can make the restrictive assumption that, since neither is said in the manual that the concurrent use of file objects can induce unpredictable results, nor that such use should be considered erroneous, I/O operations have to be considered as atomic actions, i.e., they cannot interfere with each other.
This assumption implies that the I/O operations should be modeled as (a nondeterministic choice among possible) single actions, even in the case that the manual explicitly describes their effect explicitly mentioning some sequential steps (this is the case, for instance, of the READ procedure of the DIRECT_IO package, whose definition is given in the manual in three simpler steps - the very read and two index settings). If interfering, such actions would not be allowed to be composed in parallel.

b) The second, less restrictive, interpretation is that the reference manual actually allows any kind of interference, and that the informal steps in which an I/O operation is described (e.g. the three steps of DIRECT_IO.READ) should be modelled as a sequence of elementary actions, possibly overapping when several operations are executed concurrently. In this case the effects of the execution of a single I/O operation can be rather diffrent from those expected in the sequential case. Moreover, even in the case in which an I/O operation is informally described as a single step, concurrent executions of the operation might be allowed to produce unpredictable effects (e.g. the case of the OPEN operation).

The approach adopted in the Trial Definition is to consider as atomic (i.e. indivisible and not interfering) only all those operations whose informal description does not explicitly mention a sequence of internal steps (e.g. OPEN, CLOSE, ...).
However the correctness of the adopted approach should be verified (an AJPO clarification seems needed), hence the Final Definition will not necessarily adopt the same approach.

5 IMPLEMENTATION DEPENDENCES

Most characteristics that at a first glance seem implementation dependent are on the converse dependent on the external environment (namely, the syntax and semantics of NAME and FORM, the raising of various exceptions, and so on); hence, their modeling is referred to the model of external environment, outside the scope of the present formal definition.

Moreover, we have seen that the specification as "implementation dependent" given for type

FILE_TYPE (of the generic package DIRECT_IO) means only that there is some freedom in the implementation choices, which however does not affect the abstract properties of the type itself (also due to the restrictions on the set of programs of interest, which exclude programs which directly refer the internal structure of a file object - such a program could not be legal on every implementation).

In the end the only aspects of the DIRECT_IO package that turn out to be semantically dependent on the implementation seem to be:

i) the range of the COUNT type,

ii) the consistency check of the read value with the type ELEMENT_TYPE (note that this check can also be omitted by an implementation for complex types).

This kind of implementation dependences are pointed out making parameterized the definition of the semantics of a program. With respect to the definition of the DIRECT_IO package, we need to provide as parameter the value of the upper bound of the range of the COUNT type and the test operation (part of the GLOBAL_INF specification) used to specify the global condition under which the synchronous actions READ and RAISE_DATA_ERROR are allowed.