PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

DASIM1: A PRACTICAL EXERCISE IN DATA ABSTRACTION

Geoffrey J. Nolan

Basser Department of Computer Science
University of Sydney

ABSTRACT

The program DASIM1 accepts an axiomatic specification for a data structure, then constructs an instance of the structure thus defined and executes operations upon it. This paper explains how such a specification is developed, indicating some of the problems which may be encountered. The manner in which DASIM1 employs the axioms in the simulation of the structure is described. Finally, certain correctness proofs concerning the axioms are discussed.

1. INTRODUCTION

Data Abstraction is the process of 'correctly' specifying the properties of a data structure without regard to details of implementation. We regard a 'correct' specification as one which provides all relevant details about the structure (and no irrelevant details), and which contains no ambiguities, contradictions or redundancies. Or, in model theoretic terms, we wish our specification to be complete, consistent and independent [Foo and Nolan, 1978].

Such specifications may be expressed in many different fashions, but we have chosen an equational axiom system. The construction and application of such a system is discussed in the following sections.

The program DASIM1 accepts an axiom system as input, and provides a simulation of the corresponding data structure. Simulation is perhaps not the appropriate word to use in this case, as the program actually constructs and executes operations on an instance of the structure arising from the axioms.

The program as it stands does perform various syntactic checks on the axioms as they are read. However, future versions of the program will also check the axioms for incompleteness, inconsistency, redundancy, and other semantic errors. The test for consistency (and independence) is particularly efficient because of the axiom system used by the program. This test will be described later.

The program is also limited to those structures which can be expressed using axioms of one particular form. At present, such structures consist primarily of linear and circular (i.e. non-branching) structures which are well-behaved, in the sense that the operations performed on them have the same effect, in general, on each instance of the structure.

2. CONSTRUCTING AN AXIOMATIC SPECIFICATION

Clearly, before any simulation can take place, we must specify to the program exactly the type of structure we require. DASIM1 expects this information in the form of a list of axioms, the exact nature of which will be described in the following sections.

The task of constructing an axiomatic specification is a complex one, so we will illustrate the process throughout with a sample structure. The so called 'peek-stack' [Majster, 1977] will be used for this purpose, since while being fairly simple conceptually, it poses most of the problems normally met in this area.

The peek-stack may be considered to be a normal stack equipped with a 'pointer'. The pointer can be moved down the stack by a special operation (DOWN), and reset to the top-of-stack by another (RETURN). The READ operation, instead of returning the top element of the stack, returns the element currently indicated by the pointer. Any PUSHing or POPing attempted while the pointer is not at the top-of-stack results in an error condition.

In order to establish a specification for any data structure, we must first state the operations with which we wish to equip the structure. Thus, for our peek-stack, we need the operations PUSH (an item onto the stack), POP (an item off the stack) and READ (an item). In addition, we will need the operations DOWN and RETURN described above.

Finally, we need at least one structure to act as a reference point. In this case, we choose two: NEWPS returns a new (empty) peek-stack, while ERROR is a distinguished structure which results from any illegal combination of operations. Note that we may regard NEWPS and ERROR either as constant elements of the set of peek-stacks, or as nullary functions which map into that set. Conceptually, the former definition is closer to our true intention, but we can use the same techniques on all the operations if we regard them all as functions.

Now that we have defined the operations we wish to use, we can express any peek-stack by combining these operations. For example, the peek-stack consisting of the elements '2' and '6', with '6' at the top-of-stack and the pointer indicating '2' would be expressed in the form

```
DOWN.PUSH(6).PUSH(2).NEWPS
```

The operations are performed from right to left and are separated by a dot. Henceforth, we will refer to such a string of operations as an op-sequence. The value of the above op-sequence is the peek-stack just described.

As another example, the value of

```
READ.DOWN.PUSH(6).PUSH(2).NEWPS
```

is '2'. Note that the value of an op-sequence is an element of the set mapped into by its leftmost operation (see next section).

3. FUNCTIONALITY

As stated above, the operations we wish our structure to possess may be considered as mappings. In the case of the peek-stack (and all other structures compatible with DASIM1), such mappings involve two sets. These will be referred to as D, the set of structures (in this case peek-stacks), and I, the set of items (DASIM1 uses integers).

While operations of any functionality are possible, the current version of the program recognises only the following four cases :

```
1 : D x I -> D   (PUSH)
2 : D     -> D   (POP, DOWN, RETURN)
3 :       -> D   (NEWPS, ERROR)
4 : D     -> I   (READ)
```

It is this factor which is chiefly responsible for the limitation on the type of data structure that DASIM1 is capable of representing. The reason for this restriction will become apparent later.

4. HIDDEN OPERATIONS

We have already stated that many data structures cannot be specified at all using operations with only the above four functionalities. It is also true that many structures would require an infinite number of axioms, if only the desired operations are used in the axioms. The peek-stack is such a structure [Thatcher et al., 1978].

However, by using additional operations in the axioms, finite axiom systems can be developed in some cases. The addition of an operation SHOVE (D x I -> D) is sufficient for the construction of a finite axiom system to represent the peek-stack. SHOVE has the effect of adding an item to the top of the stack irrespective of the current position of the pointer.

Of course, we may not wish the user to have access to SHOVE. It is possible within DASIM1 to specify that certain operations found in the axioms should not be available during simulation. We call such operations 'hidden' operations.

5. THE AXIOM SYSTEM

In the context of data abstraction, the term 'axiom' usually refers to a statement asserting the equivalence of seemingly dissimilar structures (i.e. op-sequences). Thus the statement "a PUSH followed by a POP is equivalent to the identity operation" could be considered an axiom. A more common form for such an axiom would be POP(PUSH(A,B)) = A .

The notation used in DASIM1 is POP.PUSH(X).$ = $, where the dollar sign is a variable representing any element of D.

The axioms which define the peek-stack to DASIM1 are :

```
PS1  : PUSH(X).#ERROR = #ERROR
PS2  : PUSH(X).#SHOVE(Y).$ = #ERROR
PS3  : #SHOVE(X).#ERROR = #ERROR
PS4  : POP.#ERROR = #ERROR
PS5  : POP.NEWPS = #ERROR
PS6  : POP.PUSH(X).$ = $
PS7  : POP.#SHOVE(X).$ = #ERROR
PS8  : DOWN.#ERROR = #ERROR
PS9  : DOWN.NEWPS = #ERROR
PS10 : DOWN.PUSH(X).$ = #SHOVE(X).$
PS11 : DOWN.#SHOVE(X).$ = #SHOVE(X).DOWN.$
PS12 : RETURN.#ERROR = #ERROR
PS13 : RETURN.NEWPS = NEWPS
PS14 : RETURN.PUSH(X).$ = PUSH(X).$
PS15 : RETURN.#SHOVE(X).$ = PUSH(X).RETURN.$
PS16 : READ.#ERROR = !E
PS17 : READ.NEWPS = !N
PS18 : READ.PUSH(X).$ = X
PS19 : READ.#SHOVE(X).$ = READ.$
```

The '#' before ERROR and SHOVE indicates the hidden nature of these operations. '!E' and '!N' are distinguished elements of I which are defined in the axioms to indicate the empty and error conditions; they are not normally available to the user during simulation. X and Y are arbitrary elements of D.

The axioms are used in the following manner. As stated earlier, each op-sequence corresponds to some peek-stack (or element of I). We apply an axiom to such a sequence by trying to match the left-hand-side (LHS) of the axiom to some right subsequence of the operations. If a match occurs, we substitute the RHS of the axiom. This produces a new string which represents the same structure. If the axioms above are repeatedly applied to any op-sequence representing a peek-stack, the sequence will be transformed into its 'Standard Form' (see below).

We will refer to an axiom system such as the one above as an NASF (Non-Associative Standard Form) equational axiom system.

The axioms are non-associative in that where more than one axiom could conceivably be applied to an op-sequence, the axiom whose LHS matches the smallest possible right subsequence of operations is always chosen. Contradictions can quite easily be forced if the axioms are applied in any other order. For example, READ.PUSH(2).#ERROR would take the value '2' if PS18 were applied first. However, the NASF system ensures that PS1 (which matches the smallest right subsequence: PUSH(2).#ERROR) is applied first, then PS16 is applied to give the correct result ('!E').

Since any given instance of a data structure can in general be represented by a number, possibly infinite, of different op-sequences, it is useful to be able to choose a unique sequence to represent each instance. We call such a sequence the 'Standard Form' (or SF) of the structure.

For example, the standard form chosen for the peek-stack is #SHOVE*.PUSH*.NEWPS + #ERROR . That is "a NEWPS followed by zero or more PUSHes, followed by zero or more SHOVEs, or an ERROR by itself".

Each axiom is of the following form :

```
OP1.OP2. ... .OPM.DS = op1.op2. ... .opN.DS
```

where OPi and opi are operations, DS is either a nullary operation or $, and the following restrictions apply :

```
(1) OP1.OP2. ... .OPM.DS  is not in SF.
(2) OP2.OP3. ... .OPM.DS  is in SF.
(3) M >= N   and   N >= 0   and   M > 0.
(4) If M = N, then op2. ... .opN.DS is not in SF.
```

(1) and (2) imply that the LHS of each axiom is of the form 'OP1.S' (where S is an SF sequence). This reflects the fact that axioms must always be applied to the smallest non-SF right subsequence.

(3) and (4) tell us that when the axiom has been applied, either SF is restored, or a new axiom can be applied to a new and smaller right subsequence.

The use of the NASF system has a number of advantages. Perhaps the most important is that the nature of the axioms follows directly from the SF. For once we have obtained the SF, we form the LHS of our axioms by prepending the operations to various SF sequences.

Another advantage of NASF is that the axioms are much more amenable to proofs of completeness, consistency etc. This is because only one axiom is normally applicable to any given op-sequence. Finally, of course, no proofs for associativity of axioms are necessary (or possible).

A possible disadvantage of the NASF system is that it may not be capable of representing certain classes of data structures. Although this possibility is mentioned in Thatcher et al. [1978], the exact extent of such restrictions is still an open problem.

6. CORRECTNESS

Once the axioms have been established, it is highly desirable to be able to prove them 'correct'. By 'correctness', we do not mean that the axioms specify the 'intended structure', since without a pre-existing specification it is impossible to determine what the 'intended structure' is. (Of course, it may be possible to prove two specifications equivalent, but that problem is beyond the scope of this discussion.)

We will describe an axiom system as 'correct' if it gives rise to exactly one type of data structure. A correctness proof can be divided into two parts. Firstly, the axiom system must contain no internal contradictions (i.e. it must be consistent). Secondly, the effect of every possible combination of operations must be clearly defined by the axiom system (i.e. it must be complete).

A third property we may wish our axioms to possess is independence. That is, no two axioms must apply to the same op-sequence. While not strictly necessary, independence is a desirable property as we avoid the complications of unnecessary axioms and simplify other proofs concerning the axioms.

Now, the NASF system requires that axioms be applied only to the smallest possible right subsequence, and that the axioms are repeatedly applied until SF is reached. Therefore, inconsistency and redundancy can only be present when two or more axioms can be applied to the same op-sequence, and hence to the same right subsequence. This condition can only occur when the LHS of one axiom is the same as, or a left subsequence of, the LHS of another axiom. Note that in this case we ignore any '$' in the LHS. By applying this simple test to all the axioms, we can determine whether any inconsistency or redundancy is present.

The second phase of the correctness proof is to ascertain that every legal op-sequence can be converted to SF by repeated application of the axioms. If this is the case, we call our axioms 'complete'. Moreover, since on any computer we can only generate finite op-sequences, we can say that for all practical purposes a complete system is also categorical (i.e. can only generate one type of structure).

Completeness proofs for individual axiom systems suggest that a testing algorithm would rely on induction on the lengths of op-sequences. However, at the time of writing, no general completeness algorithm had been perfected.

7. <u>THE PROGRAM 'DASIM1'</u>

DASIM is being written in two stages. Stage 1 accepts correct axioms as input and thereafter acts as a "black box" containing the data structure. Stage 2 (currently being designed) will accept axioms and the nature of the SF. The axioms will be checked for compatibility with the SF, and for any semantic errors (incompleteness, inconsistency etc.). Thereafter, the simulation is as before.

DASIM1 is written in standard PASCAL [Jensen and Wirth, 1975] (future versions may be written in SNOBOL4). The program accepts NASF axioms, whose operations are of the four functionalities described above. This purely arbitrary restriction ensured that the essential features of the prototype program were not submerged in the morass of implementational details associated with more general data structure simulation.

However, it is hoped that future versions of the program will accept operations of a much wider (and perhaps even arbitrary) range of functionalities. In addition, the restriction on using only the sets D and I may be lifted. This will correspondingly widen the range of data structures that DASIM is capable of representing.

A brief summary of the program's operation follows :

The program operates in two modes, an axiom gathering mode and a simulation mode. Initially, the program reads a file containing axioms written in the form :

AXNAME : LHS = RHS ; COMMENT

Each axiom is then checked for syntactic errors (such as functionality conflicts and incorrect syntax).

The axioms are then stored on a linked list, each axiom itself pointing to linked lists of operators (LHS and RHS). There is a separate list in which is stored the details of the various operations.

The axiom gathering mode is terminated by an escape character which also indicates how much of the internal workings of the structure are to be displayed.

After the axioms have been verified, the program enters the simulation mode during which it interactively accepts input lines consisting of single operations (or commands such as "clear the current structure" etc.). The program maintains a circular list on which the current SF of the structure is stored. Note that a circular list is used for convenience only, a linear list is sufficient to store circular structures. When an input line is received, the appropriate operation is added to the list. The axioms are now applied to the list as follows.

Successively larger right sublists are compared with the LHS of each axiom in turn. If a match occurs, the RHS of the axiom is substituted for the sublist. The process is repeated until no further matches can be found (i.e. SF is achieved). However, if the most recent operation returns an item, the whole structure is duplicated and the axioms are applied to the duplicate. This is because the process of extracting an item invariably has a destructive effect on the stored structure. The integer value of the operation is then displayed and the original structure is left intact.

8. THE CONCEPTUAL NATURE OF 'DASIM1'

We have variously referred to the operation of DASIM1 as 'construction' and 'simulation' of data structures. In fact, DASIM1 embeds data structures onto an internal structure, namely the linked list in which instances of the structure are stored. If we allowed the operators to have more general functionalities, they would have to be stored in a more general data structure. For example, the operations necessary for a finite axiomatisation of tree-like structures would themselves have to be internally stored on a tree (or more general structure).

Taking this concept to its logical conclusion, we should be able to simulate any data structure below a certain level of complexity by designing a suitably general internal structure for our simulating program. However, each such increase in generality entails a substantial increase in the size and complexity of the simulating program. Nevertheless, it should be possible to write programs operating on the same principles as DASIM1 for a number of general applications.

REFERENCES

FOO, N. Y. and NOLAN G. J. (1978):
"Correctness of Algebraic Specifications for Data Structures".
Proceedings Australian Universities Computer Science Seminar, Feb. 1978.

JENSEN, K. and WIRTH, N. (1975):
"PASCAL User Manual and Report" (2nd Edition).
Springer-Verlag, New York.

MAJSTER, M.E. (1977):
"Limits of the 'Algebraic' Specification of Abstract Data Types".
ACM SIGPLAN Notices, 12(10), Oct. 1977.

THATCHER, J.W., WAGNER, E.G. and WRIGHT, J.B. (1978):
"Data Type Specification: Parameterization and the Power of Specification Techniques".
Proceedings 10th STOC Conf., May 1978.