

NRL Report 8373

An Abstract Type for Statistics Collection in SIMULA

CARL E. LANDWEHR

Communications Sciences Division

March 10, 1980



NAVAL RESEARCH LABORATORY
Washington, D.C.

Approved for public release: distribution unlimited.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Report 8373	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) AN ABSTRACT TYPE FOR STATISTICS COLLECTION IN SIMULA	5. TYPE OF REPORT & PERIOD COVERED Final Report	
	6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) Carl E. Landwehr	8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Code 7522 Washington, D.C. 20375	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NRL Problem 0113-0 Task Area RR014-09-41 Program Element 61153N	
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Office of Naval Research Arlington, VA 22217	12. REPORT DATE March 10, 1980	
	13. NUMBER OF PAGES 22	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified	
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Abstract types Software Software engineering Data abstraction Programming Simulation Statistics collection SIMULA Software design		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Although the use of abstract types has been widely advocated as a specification and implementation technique, their use has often been associated with programming languages that are not widely available, and examples published to date are largely of the "toy" variety. SIMULA is a widely available language that supports the use of abstract types. The purposes of this paper are (1) to demonstrate the application of the concepts of data abstraction to a common problem; (2) to demonstrate the use of data abstraction in a widely available language; and (3) to provide a portable facility for		

(Continued)

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)

statistics collection that may make the use of SIMULA more attractive. A consistent set of terminology for discussing abstract types is presented, followed by a discussion of the background and requirements for an abstract type for statistics collection. A SIMULA implementation is given, with examples of its use. Finally, implementation of the abstract type in other languages is discussed.

CONTENTS

INTRODUCTION	1
TERMINOLOGY	2
BACKGROUND	3
REQUIREMENTS	3
SIMULA IMPLEMENTATION	5
USAGE	11
DISCUSSION	16
SUMMARY	18
ACKNOWLEDGMENTS	18
REFERENCES	18

AN ABSTRACT TYPE FOR STATISTICS COLLECTION IN SIMULA

INTRODUCTION

The use of abstract data types in the specification and implementation of programs has received much attention in the computing literature over the last several years [1-5]. Programming languages recently developed or proposed often make a point of including facilities for type abstraction [6-10]. One of the benefits of facilities of this nature should be that libraries of useful abstract types could be constructed and used in a way corresponding to the way subroutine libraries are used to compute commonly used functions. CLU [7], for example, includes a library function for just this purpose.

Although many of the articles describing language features or specification methods using abstract types include specifications for a few abstract types, these are generally in the nature of simple examples, not fully elaborated definitions of types that would be appropriate for inclusion in an application program. In addition, few of these languages are widely available.

SIMULA [11,12] includes many of the features for type specification that are advocated in more recent language designs; several authors [3,6-8] cite SIMULA as a source of ideas. There are compilers available for SIMULA on a number of widely available machines, including DEC, IBM, Univac, and Control Data mainframes. Despite its availability and the presence of some desirable features in the language, SIMULA has not achieved as widespread use in the United States as several other languages for programming and simulation that have considerably less flexibility. There are many reasons for this, not the least of which is the fact that SIMULA, compared with GPSS, GASP, and SIMSCRIPT, provides fewer built-in functions for statistics collection and reporting. Other contributing factors include the lack of any commercial organization promoting the language, lack of suitable documentation until the publication of [12], and the false perception that the language is useful only for simulation.

This report presents a set of related types defined in SIMULA suitable for the collecting and reporting of statistics in simulation programs written in SIMULA. These types represent a practical application of the concepts of data abstraction to a common problem, using a widely available programming language that includes features to facilitate the use of data abstraction. These types have been used in several simulations of queueing models and are equally applicable to statistics collection in simulations generally. Our purposes are (1) to demonstrate the application of concepts of data abstraction to a common problem; (2) to demonstrate the use of data abstraction in a widely available programming language; and (3) to provide a portable facility that may make the use of that language more attractive.

Before the types are presented, the terminology used to describe them is defined and the considerations that led to the development of these types are discussed. After defining the types, I provide examples of their use. A final section discusses problems that occurred in the development and use of these types, and notes how additional language features (some of which are included in CLU, Alphard [8], and the Ada proposals [9,10] might improve the

implementation. Information on using these types in other SIMULA programs is discussed later in this report. A general familiarity with SIMULA syntax is assumed.

TERMINOLOGY

Because there is no standard nomenclature for discussing types, variables, abstract types, etc., we must define these terms briefly. A comprehensive set of definitions was proposed in [13], and those definitions are adopted here. A *variable* is defined to be an information holder. *Access operators* supply information to or retrieve information from variables. If a variable is implemented using independently implemented variables, then those variables are referred to as the *representation* of the newly implemented variable. Thus, the access operators of a variable reference the representation of that variable. The *implementation* of a variable is the representation of that variable and the programs that provide its access operators. For any given analysis, certain *primitive* variables are not considered to have an underlying representation in terms of other variables.

An *abstract specification* of a variable is a description of the externally visible behavior of its access operators. Specifications usually distinguish behavior for legal and illegal sequences of access operator calls. A variable *satisfies* a given specification if its externally visible behavior conforms to all specified requirements for legal behavior. Two specifications are *equivalent* if any variable that satisfies one also satisfies the other. Given two specifications S_1 and S_2 , S_1 is *stronger than* S_2 (S_2 *weaker than* S_1) if any variable that satisfies S_1 also satisfies S_2 and there can be variables that satisfy S_2 but not S_1 .

A *type* is an equivalence class of variables. Two primitive variables are of the same type if their abstract specifications are equivalent. Two nonprimitive variables have the same type if their representations have the same types and if their access operators are provided by the same programs. An *abstract type* is a class of types that have common properties. For example, (1) the types may be related by a common *specification* (called *spec-types*); (2) they may have a common *representation* (*rep-types*); or (3) they may be simply listed as being in the same class (*enumerated-types*). The types discussed below are all of the first kind—they are related by a common specification, referred to as the *characteristic specification* of the abstract type.

Within the class of spec-types, three subcategories are distinguished according to the relation that the member types bear to the characteristic specification. The specifications for constituent types may be *equivalent* to the specification (*E-spec* types), weaker than the characteristic specification (*W-spec* types) or stronger than the characteristic specification (*S-spec* types). The abstract type described below is an S-spec type: the characteristic specification expresses common properties of statistics gathering, properties that do not depend on whether one is gathering statistics about real quantities, integer quantities, etc.

Descriptions of S-spec types can be classified as *independent*, *parameterized*, or *enlarged S-spec* types. Descriptions of S-spec types are independent if the member types are all independently implemented. Parameterized S-spec types are obtained when a constant in a single type definition is replaced by a parameter so that related, individual type descriptions can be obtained by associating different variables with that parameter. Enlarged S-spec types are defined by referring to existing types and giving additional representation variables and access operators. This can be done in SIMULA using prefixed classes. The abstract type I will describe is an enlarged S-spec type, but I will also discuss the use of parameterized S-spec types.

BACKGROUND

The project that led to the design of the abstract type described below was the construction of a simulator for modeling traffic flow over a broadcast channel on a communications satellite under a variety of communication protocols. The design and construction of the simulator is documented in NRL technical memoranda [14-16]. Study results for particular protocols and traffic distributions are documented in [17,18].

An important design goal was to produce software that would be easily modified to model alternative protocols and traffic distributions. To this end, it was decided to employ state-of-the-art software engineering techniques, including the use of abstract types. This decision led to the choice of SIMULA as the programming language, since it provides better facilities for user type specification than other available simulation languages (e.g., SIMSCRIPT, GPSS, GASP, FORTRAN).

Although SIMULA provides a rich structure for the definition of new types, it lacks some of the built-in facilities for statistics collection and reporting that are provided by GPSS and GASP. Palme [19] has suggested an approach that involves adding auxiliary variables and procedures for recording purposes. As an example, an integer variable would be added to the declarations for queues in the program to record the current length of the queue. Each time a new instance of a queue is created, the associated statistics collection variables are created along with it. If the programmer desires to have some queues without statistics collection, two different declarations must be provided in the program—one for queues with statistics and one for queues without statistics. These types must have different names even though they represent essentially similar objects. Finally, Palme's solution does not simplify reuse of the statistics gathering facility. Each new object type for which information is to be collected represents a new case.

Our approach is to consider the fundamental properties required of an item of statistical information—the information storage required and the operations desired—and to create a set of types suited to these requirements. The general goals are:

1. **Ease of use:** recording statistics for a new item or deleting statistics collection when no longer needed should be simple.
2. **Encapsulation of statistical calculations:** code to calculate standard statistics (e.g., mean, variance) should be located in a single place to reduce the possibility of errors and to simplify debugging.
3. **Ease of reporting:** useful reports should be easy to obtain. No elaborate format specifications should be required.
4. **Reusability:** the facilities constructed should be usable in other SIMULA programs with few or no changes.

REQUIREMENTS

For any queueing simulation, two kinds of reports are generally desired: statistical summaries of the value of some variable and history traces or histograms of the values assigned to a variable throughout the run. The statistical summaries are usually limited to observations of

the first two moments of the variable, since obtaining the required number of replications to generate statistically significant measures of higher moments is often difficult. Traces are most often used in debugging or in checking that the behavior of an indicator variable over simulated time is as expected. Histograms may be applied to similar ends or may be used to gain intuition about the general shape of a probability distribution.

In addition to supporting the collection of data for such reports, a mechanism is required to support the generation (printing) of the reports. Thus a name and description for each variable is needed, and there must be an access program that initiates the generation of a report.

The required access functions are:

- **Initialization:** the name and description of the variable to be recorded must be supplied, and the type of statistics desired (e.g., simple statistical summary, summary and histograms, trace) must be defined before collection can begin.
- **Measurement:** each time a significant change occurs in the variable to be recorded, the values of the corresponding statistics must be updated.
- **Reporting:** at the end of the run, the records must be computed and printed.

Our next step in refining these general requirements is to specify in detail the statistics desired in the reports. I chose:

1. Number of observations of the variable
2. Sum of all observations of the variable
3. Times of first and last observation
4. Minimum and maximum values observed for the variable
5. Mean and variance of the variable based on equal weight per observation
6. Mean and variance of the variable based on time-weighted observations

For the histograms, the number of bins and the bin boundaries must be specified. As in the fifth and sixth items above, the histogram may be generated weighting all observations equally or weighting each observation by the time until the next observation occurs. Traces can be handled with the histogram mechanism by using the time of occurrence as the observation and the value of the variable to be recorded as the weighting factor.

The final requirement is that the implementation of this abstract type consume a minimum of computing resources. Simulations generally are heavy consumers both of storage and of processing time, and if statistics collecting and reporting consume too many resources, the simulation may have to be restricted in other areas (e.g., fewer runs will be made or less detailed models will be necessary).

SIMULA IMPLEMENTATION

This section describes the abstract type for statistics collection as it has been implemented in SIMULA. No formal specifications were written in this project. Since it was a one person effort, the additional burden of constructing formal specifications seemed unwarranted. Consequently, the SIMULA code corresponds both to the specification and implementation of the types. Because of the structuring facilities provided by SIMULA, I think the absence of separate formal specifications is not serious.

Before describing the details of the implementation, I will give a brief example of the use of the facility. To initialize a variable for recording statistics on the number of idle servers in a queueing system, for example, one writes:

```
nidleserver:—new STATINT("Nidleserver","Number of idle servers");
```

This statement both allocates and initializes a variable for collection of integer statistics. The name and meaning of the variable to be recorded are given as arguments to STATINT so that they may be used later in the generation of reports.

To record a new observation of the number of idle servers, one writes:

```
nidleserver.update(nidle, time);
```

Here we assume that the integer variable *nidle* has as its value the current number of idle servers and that the real variable *time* records the time of the observation.

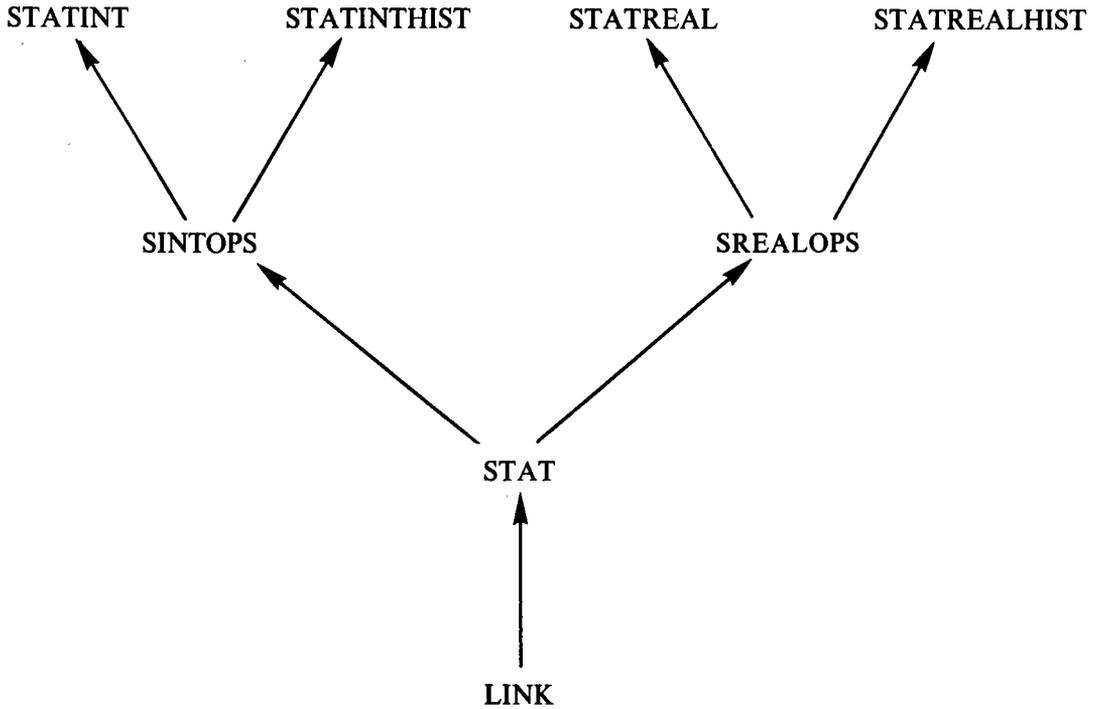
At the end of a simulation, a statistical summary for *nidleserver* is generated by the following statement:

```
nidleserver.report;
```

The report generated in this case is a two-line summary of the behavior of the variable during the simulation run, labelled with the name and description provided when *nidleserver* was initialized.

A more detailed description of the usage of the facility is given below, but these three types of statements are the primary ones required by a user. In fact, the report generation for all statistics variables can be handled by a simple loop; there is no need for a separate statement to request a report for each variable.

In the example above, *nidleserver* is an instance of the SIMULA *class* STATINT; in the terminology defined in this report, STATINT is a type. The abstract type for statistics collection in fact includes four separate types: STATINT, STATINTHIST, STATREAL, and STATREALHIST. Each of these types is defined in terms of lower level types: STATINT and STATINTHIST are both enlarged types based on another type named SINTOPS. Similarly, STATREAL and STATREALHIST are enlarged types based on SREALOPS. Both SREALOPS and SINTOPS are enlarged types based on a type named STAT. STAT is itself based on a built-in SIMULA type for linked lists called LINK. Figure 1 displays these relationships graphically.



(arrows point to the enlarged type from the type on which it is based)

Fig. 1 — Relations among types

Table 1 — Types and access functions

Type (=CLASS name)	Access functions implemented by this CLASS	Purpose
STATINT	UPDATE	simple integer statistics
STATINTHIST	UPDATE,OUTHIST	integer statistics and histograms
STATREAL	UPDATE	simple real statistics
STATREALHIST	UPDATE,OUTHIST	real statistics and histograms
SINTOPS	REPORT,EMEAN*,EVAR*	groups access functions common to integer statistics
SREALOPS	REPORT,EMEAN*,EVAR*	groups access functions common to real statistics
STAT	TMEAN*,TVAR*	groups access functions common to all statistics
LINK	INTO,OUT,PRECEDE,FOLLOW	access functions for linked list elements

*denotes access function not intended to be used directly by user programs

Table 1 lists the type names and the access functions implemented for each type. The access functions available to users of a given type include those implemented by that type and also those implemented by the type on which that type is based. Thus the access functions available for objects of type STATINT include REPORT and INTO as well as UPDATE. As the table shows, certain access functions are not intended to be called directly by users of the types. This intention could be enforced by using the HIDDEN and PROTECTED features of SIMULA, but in the case at hand (a one-person project) this added protection was not necessary.

The DEC PDP-10 SIMULA [20,21] code for the implementation of the abstract type is displayed in Figs. 2 through 9. Now I will describe the individual types starting with STAT, which is the basis for the other types.

The characteristic specification for the type STAT corresponds to the code presented in Fig. 2 for *class* STAT. This class is prefixed by *link* (a built-in SIMULA type for elements of a linked list) to allow each instance of the abstract type STAT to be linked together with the others in a single list. Thus STAT (and all other classes prefixed with *link*) are enlarged types that have the characteristic specification for *link* in common. Similarly, the variables and procedures declared in STAT are provided to all of the instances of other classes (defined below) prefixed by STAT. The declaration of procedure REPORT as *virtual* indicates that this operation can be applied to objects of type STAT but the procedure will be specified at the higher levels of classes with the STAT prefix. The reason for this construction is explained below. Procedures TMEAN and TVAR compute the mean and variance for the time-based statistics collection, and are only called from within the type. These procedures return a boolean value indicating whether a valid mean or variance could be computed. The actual mean or variance computed is left in a variable accessible to all of the procedures defined within STAT.

```

!*****;

!envelope class for all types of statistics;

!*****;
link class stat(vname,vdesc);value vname,vdesc;
  text vname;           !name of variable being observed;
  text vdesc;          !description of variable observed;
  virtual: procedure report; !procedure to print results;
begin
  integer nob; !number of observations;
  real   tfirstobs, !time of first observation;
         tlastobs, !time of last observation;
         valtint, !time integral of observed value;
         valsqtint, !time integral of square of observed value;
         timemean, !mean over time;
         timevar, !variance over time;
         eventmean, !mean over number of observations;
         eventvar, !variance over " " ";
         tinc; !temporary;
  boolean procedure tmean; !computes time-average;
  if tlastobs>tfirstobs
    then begin timemean:=valtint/(tlastobs-tfirstobs);
           tmean:=true;end
    else tmean:=false;
  boolean procedure tvar;
  if tlastobs>tfirstobs
    then begin timevar:=(valsqtint-(valtint*(valtint/(tlastobs-tfirstobs))))
           /(tlastobs-tfirstobs);
           tvar:=true;end
    else tvar:=false;
end of stat;

```

Fig. 2 — Class STAT

C. E. LANDWEHR

```

!*****;
!       envelope class for statistics collection -- integer variables;
!*****;
stat class sintops;
begin   integer val,           !initial value of variable to be logged;
        sum,                  !running sum of values;
        ssq,                  !sum of squares of values;
        max,                  !maximum value observed;
        min;                  !minimum value observed;
        boolean procedure emean;
            if nobs>0 then begin
                eventmean:=sum/nobs;
                emean:=true;end
            else emean:=false;
        boolean procedure evar;
            if nobs>1 then begin
                eventvar:=(ssq-(sum*(sum/nobs)))/(nobs-1);
                evar:=true;end
            else evar:=false;

        procedure report;      !proc to print results of simulation;
        begin integer nspaces,i,j;
            nspaces:=20-vname.length;
!code to output first line of summary statistics;

            if nspaces<=0 then outp.outtext(vname.sub(1,20))
            else begin; outp.outtext(vname);
                outp.outtext(blanks(nspaces));
                end;
            if nobs=0 then outp.outtext("    no observations recorded")
            else begin
                outp.outint(nobs,12);           !number of observations;
                outp.outint(min,12);           !minimum value observed;
                if tmean then outp.outfix(timemean,4,12)           !time average;
                else outp.outtext(" undefined ");
                if emean then outp.outfix(eventmean,4,12)           !event average;
                else outp.outtext(" undefined ");
                outp.outfix(tfirstobs,3,11);
                end;
            outp.outimage;                       !end of line 1;
!second line of summary;

            nspaces:=20-vdesc.length;
            if nspaces<=0 then outp.outtext(vdesc.sub(1,20))
            else begin outp.outtext(vdesc);
                outp.outtext(blanks(nspaces));
                end;
            if nobs ne 0 then begin
                outp.outint(sum,12);outp.outint(max,12);
                if tvar then outp.outfix(timevar,4,12)           !time variance;
                else outp.outtext(" undefined ");
                if evar then outp.outfix(eventvar,4,12)           !event variance;
                else outp.outtext(" undefined ");
                outp.outfix(tlastobs,3,11);
                end;
            outp.outimage;                       !end of second line;
            if vdesc.length>20
            then begin
                !is there more to print?;
                !yes;
                j:=20;
                for i:=21 step 20 until vdesc.length do
                begin;
                    if i+j>vdesc.length then j:=vdesc.length-i+1;
                    outp.outtext(vdesc.sub(i,j));
                    outp.outimage;
                    end;
                end;
            end;

!initialization;
nobs:=sum:=ssq:=val:=0;
min:=1000000;max:=-1000000;
tfirstobs:=tlastobs:=valtint:=valstint:=0.0;
end of sintops;

```

Fig. 3 — Class SINTOPS

```

|*****;
|   envelope class for statistics collection -- real variables;
|*****;
stat class srealops;
begin real val,          !initial value of variable to be logged;
          sum,          !running sum of values;
          ssq,         !sum of squares of values;
          max,         !maximum value observed;
          min;         !minimum value observed;
boolean procedure emean;
  if nob>0 then begin
    eventmean:=sum/nob;
    emean:=true;end
          else emean:=false;
boolean procedure evar;
  if nob>1 then begin
    eventvar:=(ssq-(sum*(sum/nob)))/(nob-1);
    evar:=true;end
          else evar:=false;

  procedure report;      !proc to print results of simulation;
  begin integer nspaces,i,j;
    nspaces:=20-vname.length;
!code to output first line of summary statistics;

    if nspaces<=0 then outp.outtext(vname.sub(1,20))
      else begin; outp.outtext(vname);
        outp.outtext(blanks(nspaces));
      end;
    if nob=0 then outp.outtext("  no observations recorded")
      else begin
        outp.outint(nob,12);          !number of observations;
        outp.outfix(min,4,12);        !minimum value observed;
        if tmean then outp.outfix(timemean,4,12)          !time average;
          else outp.outtext(" undefined ");
        if emean then outp.outfix(eventmean,4,12)          !event average;
          else outp.outtext(" undefined ");
        outp.outfix(tfistobs,3,11);
      end;
    outp.outimage;          !end of line 1;
!second line of summary;

    nspaces:=20-vdesc.length;
    if nspaces<=0 then outp.outtext(vdesc.sub(1,20))
      else begin outp.outtext(vdesc);
        outp.outtext(blanks(nspaces));
      end;
    if nob ne 0 then begin
      outp.outfix(sum,4,12);outp.outfix(max,4,12);
      if tvar then outp.outfix(timevar,4,12)          !time variance;
        else outp.outtext(" undefined ");
      if evar then outp.outfix(eventvar,4,12)          !event variance;
        else outp.outtext(" undefined ");
      outp.outfix(tlastobs,3,11);
    end;
    outp.outimage;          !end of second line;
    if vdesc.length>20
      then begin
        !is there more to print?;
        !yes;
        j:=20;
        for i:=21 step 20 until vdesc.length do
          begin;
            if i+j>vdesc.length then j:=vdesc.length-i+1;
            outp.outtext(vdesc.sub(i,j));
            outp.outimage;
          end;
        end;
      end;
    end;

!initialization;
nob:=sum:=ssq:=0;
min:=100000;max:=-100000;
tfistobs:=tlastobs:=valtint:=valstint:=val:=0.0;

end of srealops;

```

Fig. 4 — Class SREALOPS

C. E. LANDWEHR

```

!*****;
!class for recording integer variables with histograms;
!*****;
sintops class statinhist(nbins,htype,lowerbd,inc);
  integer nbins;          !number of bins for histogram;
  integer htype;         !if =htime, weight each obs by time since last obs;
                        !if =hevent, all observations count equally;
                        !if =httrace, record time as observation and value as weight;
  integer lowerbd,      !bound of first bin;
  inc;                  !increment for succeeding bins;
begin  real array a(1:nbins+1),b(1:nbins);    !a must be one element longer than b;
      !histogram is generated in a, bins are defined by b;
  procedure update(newval,tobs); !routine to be called for new observation;
    integer newval;          !the new value observed;
    real tobs;              !time of the observation;

    begin;
      nobs:=nobs+1;
      if nobs=1 then tfirstobs:=tobs;
      sum:=sum+newval;
      ssq:=ssq+newval*newval;
      tinc:=val*(tobs-tlastobs);
      valtint:=valtint+tinc;
      valsqtint:=valsqtint+val*tinc;
      if max<newval then max:=newval;
      if min>newval then min:=newval;
      if htype=hevent then histo(a,b,newval,1)
        else if htype=htime then histo(a,b,newval,tobs-tlastobs)
        else if htype=httrace then histo(a,b,tobs,newval);
      tlastobs:=tobs;
      val:=newval;
    end of update;
  procedure outhist;
  begin
    phist(vname,vdesc,nbins+1,a,b,htype); !print hist with proc, give real arrays size;
  end;
  begin integer i;          !initialization of bins;
    for i:=1 step 1 until nbins do b(i):=lowerbd+(i-1)*inc;
  end;
end of statinhist;

```

Fig. 5 — Class STATINHIST

```

!*****;
! class for recording statistics of integer variables without histograms;
!*****;
sintops class statint;
begin  procedure update(newval,tobs); !routine to be called for new observation;
    integer newval;          !the new value observed;
    real tobs;              !time of the observation;

    begin;
      nobs:=nobs+1;
      if nobs=1 then tfirstobs:=tobs;
      sum:=sum+newval;
      ssq:=ssq+newval*newval;
      tinc:=val*(tobs-tlastobs);
      valtint:=valtint+tinc;
      valsqtint:=valsqtint+val*tinc;
      if max<newval then max:=newval;
      if min>newval then min:=newval;
      tlastobs:=tobs;
      val:=newval;
    end of update;
end of statint;

```

Fig. 6 — Class STATINT

There are two types defined with prefix STAT: SINTOPS and SREALOPS (Figs. 3 and 4). As the names imply, SINTOPS provides the storage and operations for integer variables for which statistics are desired, and SREALOPS provides the corresponding services for real variables. (SIMULA defines types for real and integer variables much as they are in ALGOL or FORTRAN.) These operations cannot be provided directly at the STAT level because, for example, the value of the minimum observation of a real (integer) variable must be stored in a real (integer) variable. Thus, essentially identical sets of operations are provided with differences only in the types of the variables. The REPORT procedures for reporting summary statistics are provided at this level for the same reason: the output format requirements are different for integer and real variables.

There are two types defined with prefix SREALOPS and two with prefix SINTOPS. The classes STATINTHIST (Fig. 5) and STATINT (Fig. 6) are prefixed by SINTOPS, and STATREALHIST (Fig. 7) and STATREAL (Fig. 8) are prefixed by SREALOPS. The division in this case is between variables for which histograms are to be collected and reported and those for which only statistical summaries are required. This division is motivated by the final requirement discussed previously in the Requirements section: that the implementation of the type require as few resources as possible. Since the storage needed for the data collected to generate a histogram or trace considerably exceeds that required for generating a simple statistical summary, different types are defined for the two cases. The update operations are provided at this level since the definitions of what statistics to save depends both on the type of the variable being recorded and on whether a histogram is to be generated or not. The built-in SIMULA function HISTO is used to record data for histograms. STATINTHIST and STATREALHIST contain special operations for generating histograms; in both cases, the operation OUTHIST simply calls a global histogram printing routine, PHIST (Fig. 9), to do the actual output. This routine was made global to avoid duplicating the code for it within the STATINTHIST and STATREALHIST declarations.

USAGE

To employ these types in a simulation is straightforward. For each variable about which statistics are to be collected, a statistics collection variable must be declared, allocated, and linked into the statistics pool (Fig. 10a). At each place the value of the variable is to be recorded, the update operation must be invoked (Fig. 10b). At the end of the run, the report operation can be used to generate the statistical summary, and, if a STATINTHIST or STATREALHIST variable was created, the OUTHIST operation will print a histogram when it is called. If all statistics variables are linked in a single pool, a simple iteration can be written to sequence through them, generating a report for each. Histograms can be generated in a similar fashion, although it must be verified that each variable is of the appropriate type. Figure 11 displays a code loop that generates all of the statistical output for a simulation run.

Occasionally, a user may wish to get a histogram for a variable that was not previously recorded in this fashion. The only changes required to accomplish this are to alter the declaration slightly and to expand the initialization statement to include the histogram-dependent information, such as bin size, number of bins, lower bound, etc. (Fig. 12). No changes in the update instructions or printing routines are required. Conversely, to save the storage occupied by a histogram no longer of interest, the user merely replaces the histogram declaration and initialization statements with their simpler counterparts.

C. E. LANDWEHR

```

!*****;

!class for recording real variables with histograms;

!*****;
$realops class statrealhist(nbins,htype,lowerbd,inc);
    integer nbins;           !number of bins for histogram;
    integer htype;          !if =hevent, all observations count equally;
                           !if =htime, weight each obs by time since last obs;
                           !if =httrace, record tobs as observation weighted by newval;
    real lowerbd;           !bound of first bin;
    real inc;               !increment for successive bins;
begin
    real array a(1:nbins+1),b(1:nbins);    !a must be one element longer than b;
    !histogram is generated in a, bins are defined by b;
    procedure update(newval,tobs); !routine to be called for new observation;
        real newval;           !the new value observed;
        real tobs;             !time of the observation;

        begin;
            nobs:=nobs+1;
            if nobs=1 then tfirsttobs:=tobs;
            sum:=sum+newval;
            ssq:=ssq+newval*newval;
            tinc:=val*(tobs-tlasttobs);
            valtint:=valtint+tinc;
            valsgtint:=valsgtint+val*tinc;
            if max<newval then max:=newval;
            if min>newval then min:=newval;
            if htype=hevent then histo(a,b,newval,1.0)
                else if htype=htime then histo(a,b,newval,tobs-tlasttobs)
                else if htype=httrace then histo(a,b,tobs,newval);
            tlasttobs:=tobs;
            val:=newval;
        end of update;
    procedure outhist;
    begin
        phist(vname,vdesc,nbins+1,a,b,htype);    !use proc to gen hist (give real array size);
    end;
    begin integer i;           !initialize bins;
        for i:=1 step 1 until nbins do b(i):=lowerbd+(i-1)*inc;
    end;
end of statrealhist;

```

Fig. 7 — Class STATREALHIST

```

!*****;

!class for recording real variables, no histogram;

!*****;
$realops class statreal;
begin
    procedure update(newval,tobs); !routine to be called for new observation;
        real newval;           !the new value observed;
        real tobs;             !time of the observation;

        begin;
            nobs:=nobs+1;
            if nobs=1 then tfirsttobs:=tobs;
            sum:=sum+newval;
            ssq:=ssq+newval*newval;
            tinc:=val*(tobs-tlasttobs);
            valtint:=valtint+tinc;
            valsgtint:=valsgtint+val*tinc;
            if max<newval then max:=newval;
            if min>newval then min:=newval;
            tlasttobs:=tobs;
            val:=newval;
        end of update;
end of statreal;

```

Fig. 8 — Class STATREAL

```

!*****;
! procedure to print histograms;
!*****;
procedure phist(vn,vd,nb,a,b,htype); value vn,vd;
  text vn,vd;          !variable name and description;
  integer nb;          !number of elements in a;
  real htype;          !type of histogram this is;
  real array a,b;      !a contains histogram counts, b gives bin boundaries;
begin integer nx,i,j; !temporaries;
  integer hwidth;      !histogram width;
  real nscale;         !scale factor;
  real xmax;           !largest data value;
  hwidth:=50;          !set width of histograms;
  xmax:=0.0;
  for i:=1 step 1 until nb do if a(i)>xmax then xmax:=a(i);
  if htype=hevent then nscale:=entier(xmax/hwidth)+1
    else if htype=htime or htype=htrace then
      nscale:=xmax/hwidth;
  outp.outimage;eject(1);
  outp.outtext("Histogram for variable ");outp.outtext(vn);outp.outtext(":");outp.outimage;
  outp.outtext(vd);outp.outimage;
  outp.outimage;
  if nscale=0 then outp.outtext("No observations recorded.")
  else begin
    outp.outtext(" each x = ");outp.outfix(nscale,4,12);
    if htype=hevent then begin
      outp.outtext(" observation"); if nscale=1 then outp.outtext(".") else outp.outtext("s.");
    end;
    outp.outimage;
    outp.outimage;
    if htype=hevent or htype=htime then outp.outtext(" upper")
      else if htype=htrace then outp.outtext(" time period");
    if htype=hevent then outp.outtext(" number ");
    outp.outimage;
    if htype=hevent or htype=htime then outp.outtext(" bound")
      else if htype=htrace then outp.outtext(" ending");
    if htype=hevent then outp.outtext(" obs ")
      else if htype=htime then outp.outtext(" frequency")
      else if htype=htrace then outp.outtext(" value");
  outp.outimage;
  for i:=1 step 1 until nb do
    begin
      if i<nb then outp.outfix(b(i),4,12)
        else begin outp.outchar('>');outp.outfix(b(nb-1),4,11);end;
      outp.outfix(a(i),4,12);
      outp.outchar(' ');
      nx:=entier(a(i)/nscale);
      for j:=1 step 1 until nx do outp.outchar('X');
      outp.outimage;
    end;
  end;
  outp.outimage;outp.outimage;outp.outimage;
end;

```

Fig. 9 — Procedure PHIST

```

ref (head) statpool;
.
.
statpool:-new head;
} appears once per simulation

ref (statint) nmsgbacklog;           !declaration for statistics
                                     collection variable;

nmsgbacklog:-new statint ("nmsgbacklog","Number of messages not yet
                           transmitted"); !allocation and initialization of
                                     same variable;

nmsgbacklog.into(statpool);         !linking of variable into
                                     statistics pool;

```

a. Declaration, allocation and initialization, and linking into statistics pool

(code to generate a new message and queue it)

```

nmsgbacklog.update(nmsgbacklog.val+1,time); !record the incremented
                                             value as observed at the
                                             current time;

```

b. Example of recording an observation (Note that in this case, the statistics variable itself is used to record the value of the backlog.)

Fig. 10 — Declaration, initialization, and use of type for collection of simple summary statistics

```

!Print reports for all variables in the statistics pool. (First print ;
!statistical summaries for all variables and then histograms for those ;
!variables that are of type statrealhist or statinthead.) ;

outp.outtext("Simulation Statistics");      !print general heading (outp has
!print general heading (outp has
been previously declared and
initialized as a SIMULA printfile);
outp.outimage;outp.outimage;              !skip two lines;

outp.outtext(" Variable          # obs      minimum   time mean   event
mean  first obs");                          !first line of heading;

outp.outtext(" sum              maximum   variance   varia
nce   last obs");                          !second line of heading;

outp.outimage;outp.outimage;              !force out header and skip;

svar:--statpool.first;                    !get first statistics variable
                                           (svar previously declared
ref(stat) );
while svar /= none do                      !this loop prints summaries only
  begin      svar.report;                  !writes the report for this
                                           variable;
            outp.outimage;                !skip a line;
            svar:--svar.suc;              !get next statistics variable;
  end;

outp.linesperpage(-1);                    !suppress page skips for
                                           histograms;

svar:--statpool.first;                    !re-initialize to generate all
                                           histograms together;

while svar/=none do
  begin
  inspect svar  when statinthead do svar qua statinthead.outhist
                when statrealhist do svar qua statrealhist.outhist;

!the above statement checks that the current svar is of a histogram
type and (if so) prints it with the operation for that type;

svar:--svar.suc;                          !get the next one;
end;

```

Fig. 11 — Output generation for statistical variables of all types

```

ref (statinthist) nmsgbacklog;           !declaration for integer
                                         statistics variable with histogram
                                         generation;

nmsgbacklog:-new statinthist ("nmsgbacklog","Number of messages not yet
                               transmitted",20,htime,0,1);

!allocation and initialization for a histogram with 20 bins, using
time-weighted observations (htime is a global integer variable with
value 2), lower bound of the first bin is 0, and the increment for
each bin is 1.;

nmsgbacklog.into(statpool);             !link variable into statistics
                                         pool;

a. Declaration, allocation, and initialization of a variable to
collect both statistical summaries and a histogram.

nmsgbacklog.update(nmsgbacklog.val+1,time); !record observation;

```

- b. Example observation of variable that collects both summary information and histogram. (see Fig. 10b)

Fig. 12 — Example of declaration, allocation, initialization, and use of a variable to collect both a statistical summary and a time-weighted histogram

DISCUSSION

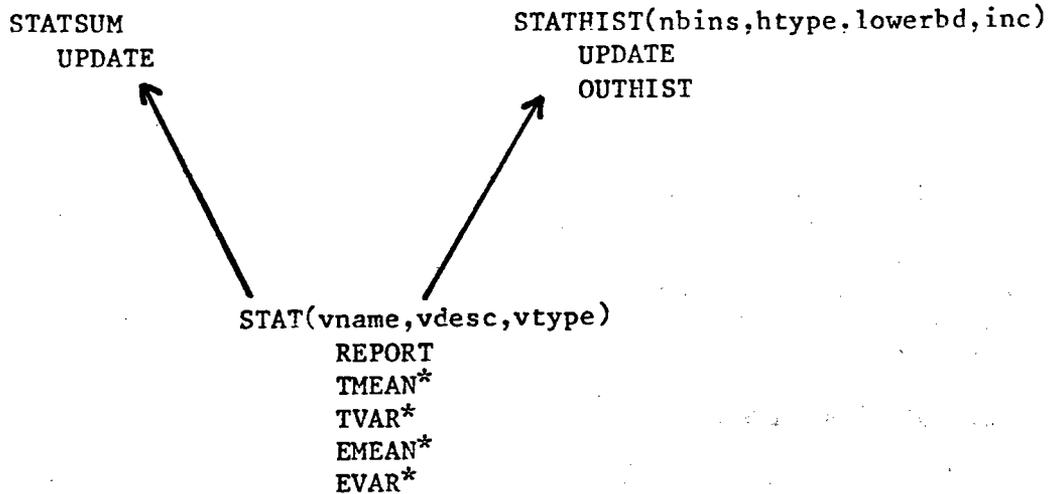
The abstract type just presented has been used without change in three versions of one simulator and in two separately developed simulators. It provides a useful facility in its present form. Experience indicates that the separation of histogram generation and simple statistical summaries is valid—if storage becomes tight in the simulation, a good deal can be saved by altering variables for which histograms had been generated to collect statistical summaries only. There are, however, some deficiencies in the SIMULA facilities for type specification that are underlined by this example.

The most noticeable of these is the inability to allow the type of a parameter itself to be parameter in a *class* definition. If this were possible, there would be no need to separate the SREALOPS and SINTOPS classes; the same code could be used for both and would only need to appear once. The abstract type could then be constructed as a parameterized S-spec type instead of an enlarged S-spec type. The operations presently implemented in SREALOPS and SINTOPS would be placed in STAT and a parameter would be added to STAT to specify the type (integer or real) of the variable about which statistics would be collected. In fact, the

STATREAL and STATINT classes could also be merged in such an environment, since, again, the only difference in the code between the two classes is in the types of the variables referenced.

In the terminology defined above, SIMULA only partially supports the specification of parameterized S-spec types. Such types can be specified where the values of a parameter in a CLASS specification are used to distinguish different member types, but types in which the parameter type itself varies among members of the abstract type can only be specified (as I have done) with enlarged types, using the class prefixing mechanism.

The difference between statistics collection variables that allow reporting of statistical summaries and those that allow histogram generation now appears to be the only substantive one. A tree that outlines a revised type structure based on preserving only this distinction is shown in Fig. 13. Notice that this revision combines STATINTHIST and STATREALHIST. This combination implies that PHIST no longer need be a globally defined subroutine—it can be included in its natural place as an operation on variables of type STATHIST without requiring two copies of the same code.



*denotes access functions not intended to be called directly by user programs
(arrows point to enlarged type from the type on which enlarged type is based)

Fig. 13 — Revised type structure

STAT now includes an argument (vtype) to distinguish whether statistics are to be recorded for a real or an integer variable. This parameterization could allow all of the operations listed to be coded only once and would eliminate the level introduced by SREALOPS and SINTOPS. STATSUM would define a single version of the update operation for variables only requiring statistical summaries, and STATHIST would include storage and operations to record (via update) and print (via outhist) histograms as well.

In the revised structure, STAT specifies a set of types, with one member for each possible value of vtype. STATSUM and STATHIST are enlarged types based on STAT (instantiated with a given parameter value for vtype). The principle advantage of the revised structure is the elimination of several nearly duplicate sections of source code required by the constraints of SIMULA. Nearly all of the code in Figs. 3, 6, and 7 could be removed. This revision would, it appears, be feasible in either of the Ada languages described in [9, 10] through use of the overloading and generic capabilities. In CLU [7], parameterized clusters might be used in the implementation. Alphard [8] also includes mechanisms that could be used to implement the revised structure.

Despite the advantages that the newer languages have in the implementation of such a facility, the user interface presented by the statistics collection type in the newer languages would probably not be substantially different from that of the present facility. Nor would the revised version be likely to require less computing time per call (although storage requirements might be slightly decreased). Viewed in this way, and considering the relatively wide availability of SIMULA, I believe that the abstract type presented above should be helpful in constructing simulation programs for some time to come.

SUMMARY

In the preceding sections, we have displayed an abstract type for statistics collection in SIMULA. The requirements for the type, the programs implementing it, and the use of the type in a SIMULA program have been presented and discussed. The implementation has been described in a consistent nomenclature, and limitations in SIMULA that prevent use of a more desirable implementation strategy have been noted in the same nomenclature. Despite these limitations, the use of abstract types for software design and implementation in conjunction with the mechanisms provided by SIMULA has proven to be a useful technique in program construction.

ACKNOWLEDGMENTS

Several colleagues at the Naval Research Laboratory have helped in the preparation of this report. In particular, the author is indebted to Drs. J. Shore and D. Parnas for the nomenclature defined in the Terminology section and used throughout the paper. Dr. Shore also provided a thorough review of earlier drafts of the paper, as did D. Weiss and Drs. D. Baker and J. Gannon.

REFERENCES

1. K. Gries and N. Gehani, "Some ideas on data types in high level languages," *Comm. ACM* 20, 6 414-420 (June 1977).
2. J.V. Guttag, "Abstract data types and the development of data structures," *Comm. ACM* 20, 6 396-404 (June 1977).
3. J.V. Guttag, E. Horowitz, and D.R. Musser, "Abstract data types and software validation," *Comm. ACM* 21, 12 1048-1064 (Dec. 1978).
4. B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Notices* 9 50-59 (April 1974).

5. D.L. Parnas, J.E. Shore, and D.M. Weiss, "Abstract types defined as classes of variables," NRL Report 7998, April 1976.
6. C.M. Geschke, J.H. Morris, and E.H. Satterthwaite, "Early experience with Mesa," *Comm. ACM* 20, 8 540-553 (Aug. 1977).
7. B. Liskov, A. Snyder, R. Atkinson, and C. Schiffert, "Abstraction mechanisms in CLU," *Comm. ACM* 20, 8 564-576 (Aug. 1977).
8. W.A. Wulf, R.L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Trans. on Software Eng. SE-2*, 4 253-265 (Dec. 1976).
9. Green Programming Language Reference Manual, Honeywell, Inc., and Cii Honeywell Bull, March 1979.
10. J. Nestor and M. Van Deusen, "Red Language Reference Manual," Intermetrics, March 1979.
11. G. Birtwistle, O.J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA Begin*, Auerbach Publishers, Philadelphia, Pa., 1973.
12. W.R. Franta, *A Process View of Simulation*, Elsevier, North Holland, N. Y., 1977.
13. D.L. Parnas and J.E. Shore, "Language facilities for supporting the use of data abstractions in the development of software systems," NRL Internal Report, February 1978.
14. C.E. Landwehr, "On the design of a simulator for satellite communications," NRL Technical Memorandum 5403-85, March 1977.
15. C.E. Landwehr, "Construction and validation of the Satellite Communications Simulator," NRL Technical Memorandum 5403-259, June 1977.
16. C.E. Landwehr, "SIMULA and events," NRL Technical Memorandum 7503-113, April 1978.
17. C.E. Landwehr, "Performance studies of the distributed CPODA protocol in the Mobile Access Terminal network," NRL Memorandum Report 4084, September 1979.
18. M. Melich, C.E. Landwehr, and P. Crepeau, "Analysis of alternative satellite channel management systems," NRL Report, to appear, winter, 1980.
19. J. Palme, "Putting statistics into a SIMULA program." Available as NTIS PB-243 785, July 1975.
20. S. Arnborg, O. Bjorner, L. Enderin, E. Ergstrom, R. Karlsson, M. Ohlin, J. Palme, I. Wennerstrom, and C. Wihlborg, *Decsystem 10 SIMULA Language Handbook, Part II*, Dec. 1974. Available as NTIS PB-243 065.
21. G. Birtwistle, and J. Palme, *Decsystem 10 SIMULA Language Handbook, Part I*. Available as NTIS PB-243 064, Sept. 1974.

