SYSTEM DESCRIPTION METHODOLOGIES
D. Teichroew and G. David (Editors)
Elsevier Science Publishers B.V. (North-Holland)
© IFIP, 1985

THE PROGRAMMER-CLIENT INTERACTION IN ARRIVING AT PROGRAM SPECIFICATIONS: GUIDELINES AND LINGUISTIC REQUIREMENTS

Daniel M. Berry[1], Orna Berry[2]
Computer Science Dept. UCLA Los Angeles, CA 90024
Computer Science Dept. USC Los Angeles, CA 90089

Abstract: This paper describes an experience by the authors as a programmer and client respectively. The programmer applied the concepts of abstract data typing, and strong typing to arrive at complete, consistent specifications of the client's program. Amazingly, these specifications did not undergo the usual modifications as the program itself was designed, implemented, tested, debugged, and accepted. The experience suggests a possible methodology for arriving at specifications based on the two above-mentioned concepts and suggests some properties of specification languages and their processors.

1. Introduction

Most software is produced by professional programmers to meet the requirements of a client who is generally not a programming or computing professional. All too often the resultant software is not what the client wants. The programmer may have misunderstood what the client wanted and produced something entirely different. The client may not have known what s/he wanted, and the software, though done correctly with respect to the specifications, does too little, too much, or the wrong thing.

Assuming for the moment that the programmer is perfect and programs correctly from the specifications, still it is extremely difficult to arrive at specifications that specify exactly what the client wants. Thus, *the major problem* in the production of software meeting the client's requirements is in obtaining mutually satisfying specifications -- specifications which specify exactly what the client wants, which perhaps even anticipate future desires, and from which the programmer may write the required software.

The difficulties preventing sufficient mutual understanding to arrive at the specifications are that the programmer, on one hand, generally knows little or nothing about the client's discipline and the client, on the other hand, knows little or nothing* about what is possible with a computer.

These authors firmly believe that the responsibility for insuring mutual understanding and getting the specifications right lies squarely with the programmer. Therefore, what are needed are means for the programmer to learn exactly what the client means, for the programmer to assist the client in defining his/her problem, and for the programmer to assist the client in learning what is possible with a computer and what s/he should be expecting from the software, etc.

Recently, the first author, a computer scientist, and the second author, at the time a statistician, entered into a programmer-client relationship in order to produce specifications for a statistical experiment simulation program. The computer scientist in this case knows very little about probability and statistics, and worse than that, has an aversion to the subject. The statistician in this case knew very little about computing beyond the use of packages such as BMDP [DB77] and beyond the barest rudiments of FORTRAN *coding* (as opposed to programming). Finally, the two authors are married to each other.

1. This work was supported in part by the Lady Davis Foundation, The Hebrew University, and the Weizmann Institute of Science in Israel and the U.S. Dept. of Energy contract No. EY-76-5-034, PA 214.

2. This work was supported in part by Tel Aviv University in Israel.

*Even worse, the client may know only a little bit from his home computer, but think s/he knows more.

In spite of this clear potential for a lack of mutual understanding, the specifications produced before programming began ended up being an accurate description of the final program as written. The specifications, to the first author's utter amazement, withstood all of the usual feedback that takes place during program development. That is, the specifications remained a useful guide throughout the programming; no new desired functions were discovered; and no specified functions were changed or eliminated.

This history is quite different from the usual history in which the specifications change significantly as the software is developed in order that the specifications agree with the final software. Inconsistencies in the specifications are found, functions are found to be not implementable, new or modified functions are found, etc., all of which force changes to the developing software and the specifications. These changes are not desirable because they delay the completion of the software and they decrease the reliability of the resulting software. The major structure of the software is designed using the original specifications; for economic reasons, the changes are generally accommodated by patches to this structure rather than by a redesign of this structure.

It is believed that what made the difference in this case was the combined use of abstract data typing [LZ74, Flo75, Par72, Mye78], strong typing [vWn75, DOD78] and a strong dosage of persistence by the programmer.

Abstract data typing is normally used to hide the details of implementing a particular data object and its operations, i.e., for *information hiding*. In this case, abstract data typing provided a framework in which to take the various statistical buzzwords (from the programmer's point of view they are buzzwords), e.g., *standard error*, and treat them as operations of a particular abstract type, the *observation vector*. In this way, without knowing what the various buzzwords mean, the programmer was able to write high level statements using the buzzwords as operations on observation vectors. In other words, abstract data typing was used in this case to hide the programmer's utter *ignorance* about how to implement the data object and its operations. Of course, the programmer pressed the client for assurance that each such buzzword is a well-known operation of statistics and that each is defined completely by some formula. Thus, the programmer was confident that the abstract type and thus the program would be implementable. Further, the abstract type provided a vehicle for the (experienced) programmer to press the client for additional functions that she did not think of at first, but that later turned out to be necessary.

By taking advantage of the notion of *strong typing*, and the compile-time type checking it permits, inconsistencies in the specifications were caught and eliminated at the source before they could do difficult-to-repair damage to the program.

*Persistence** was necessary in order for the programmer to keep pressing the client for more details and to spot inconsistencies in the stated specifications.

This report first proposes a general methodology for arriving at specifications which are mutually satisfying to the client and the programmer and which has a resistance to changes induced by feedback obtainable during the programming process. It then details the above described programmer-client interaction. It attempts to analyze why the interaction was so successful in obtaining good specifications. Finally, it makes some recommendations concerning specification languages, observing that Ada^{TM} [ADA81] comes close to meeting these recommendations and that the program design language SDP [Lin80, LYB81] comes even closer.

---

*In an earlier version of the paper and in the oral presentation, the term "Jewish Motherhood" was used instead of "persistence". Previous referees indicated that this original term may not be universally understood. However, we believe that "Jewish motherhood" connotes much more than "persistence", and some of these additional connotations are intended. What we require from Jewish motherhood are the *instincts* for the programmer to keep *nudging* the client for more details and to *sense* inconsistencies in the stated specifications. The nuances in the previous sentence are not quite captured by "persistence". Lest the reader worry that he or she cannot qualify as a Jewish mother, as it is stated in [Gre64], "You don't have to be either Jewish or a mother to be a Jewish mother. An Irish waitress or an Italian barber could also be a Jewish mother." To this list we add: "Also a (your ethnicity) programmer can be a Jewish mother."

^{TM} Ada is a trademark of the U. S. Department of Defense (AJPO).

2. Proposed Methodology for Arriving at Specifications

The experience described in the next section suggests a methodology for arriving at specifications, which are mutually satisfying to the client and the programmer and which does not require the programmer to be an expert in the client's field. The methodology requires that the programmer be experienced in the use of abstract data typing and strong typing in his or her own programming.

In essence, as the client is talking, the programmer should listen for the abstract data types relevant to the problem and try to identify their operations. The types and their operations should be clear in the client's own mind and be intelligible to anyone in his or her field. After some persistent questioning by the programmer in order to verify that the operations are in fact implementable (not necessarily in a small routine), the specifications should be written jointly by the programmer and the client. These specifications should consist of modules describing the abstract types and the modules for the main programs. The module for an abstract type should give only the name of the abstract type and the names, parameter types, and return values types for the operations of the abstract type. The main program modules should be written with the operations of the abstract types assumed as primitive.

The programmer should then use type checking and whatever other tools he or she can muster to detect inconsistencies and to press the client for answers which clear them up.

The specifications including those of the abstract data types should be circulated among the colleagues of both the client and programmer for additional feedback and checks that the specifications are meaningful to people in the client's field and that the specifications are in fact implementable. If this circulation results in any changes, the cycle of checking and circulating should be repeated. When finally all are satisfied, the programmer may begin the next step in the program's life cycle.

The methodology stresses getting the programmer and client to transfer *some* of their individual knowledge to each other rather than making the programmer an independent specialist in the client's area who is able to solve any programming problem in the area or making the client a competent programmer. The knowledge that the client gives is about the problem area, and the knowledge that the programmer gives is about what computers are able to do. The extent of this information transfer is sufficient only to permit mapping the client's needs into some representation from which the programmer may then produce a program meeting the client's needs. The transfer involves the client telling the programmer things, the programmer finding inconsistencies, and the programmer questioning the client in order to resolve these inconsistencies. In doing this, the programmer makes use of syntactic and semantic rules of natural language, instinct, and syntactic rules of the specification language. It is a contention of this paper that this use of syntactic rules of the specification language goes a long way to making up for the programmer's lack of knowledge in the client's area.

The above description of the methodology may seem vague and ill-defined, but in fact, it is as complete as possible. The actual application of the methodology is quite problem-dependent, so at best only the outlines of methodology can be given. In case the reader cannot see how to apply the methodology to real problems, the next section describes the authors' actual experience. It is somewhat anecdotal, but it does show what is involved in the use of the methodology. The experience also gives the basis for the claims made in this paper on the efficacy of the methodology and for the recommendations given later on the nature of specification languages.

3. The Experience

3.1 Background

The reader is already aware of the differing professional backgrounds of the programmer and the client and the clear potential for communication problems*. Some additional elements of background must be explained

* Add to this the fact that their native languages are different.

At the time of the experience, during the spring of 1979, the second author was an M.A. candidate in the Department of Statistics at Tel Aviv University. Her thesis [Bry79] research considered the validity of conclusions drawn from experiments involving a sequence of observations in which the data becomes further unavailable (i.e., truncated) at some point or in which some but not all of the data are missing (i.e., censored). It was desired to write a program which permitted numerous simulations of experiments with large observation vectors. An experiment was to generate a pair of vectors of observations, to truncate these and then to censor these later. Conclusion drawn from the truncated and censored vectors were to be compared to those drawn from the full vectors. Drawing the conclusions and comparing them requires the calculation of a variety of well-known statistical measures at all levels of the experiment.

The program had to meet the approval of the second author's three-person thesis committee which could, in principle, think of new calculations to perform at any time, even at the defense. The entire thesis, including the program, several production runs of it, documentation for it, the writing of the thesis and the oral defense had to be finished within ten weeks**, extended in the end by one week, from the date it was determined to begin writing the program. Thus, it was critical that the specifications be correct and complete the first time around and that they anticipate, from the outset, all possible calculations that could ever be reasonably (or even unreasonably) desired by a statistician-thesis committee member. There was simply no time to write the program more than once.

The first author was personally familiar with patchwork programs resulting from thinking of new calculations on the data during or after completion of the programming. He was also personally familiar with the agonizing discovery of a newly required calculation requiring a major rewrite of a nearly completed or completed program. The first author was painfully familiar with the discovery of interface problems among procedures late in the programming whose correction required a major rewrite or extensive patching. These situations had to be avoided at all costs.

3.2 Form of the Specifications

It was decided by the programmer with the client's concurrence that the specifications would consist of two main parts:

1. in the form of typed variable declarations together with assertions as comments, a specification of all inputs to the program and the validity constraints they have to meet.

2. given input meeting the input specifications of part 1, a specification of what the output should look like, i. e., what values, tables, etc. should be printed.

No details other than those needed to describe these parts would be given. In particular no data which needed to be calculated internally but which would not get printed out would be specified.

3.3 Initial Specifications

The programmer asked the client to write a first pass at each part of the specifications. For the input part, the following list of "declarations" was produced:

input:

N	(Sample size) INT
Θ	(parameter of exponential distribution) REAL
p	(the percentage of the data to be lost) REAL
t	(the truncation of the survey) REAL

For the output part, the following list of what was to be printed was produced:

output:

** Her advisor was leaving the country for the summer.

1. *observations;* 2 vectors of size N, each obs is a REAL no.
2. new observation vectors without the lost data (no info as to which positions lost data).
3. the 2 vectors of truncated obs.

for each of 1,2,3

4. *P.25 P.50 P.75* | of combined data and of
5. *P(t=1/2), P(t=1), P(t=2)* | each vector separately
6. table of nonparametric test
   each one of elements of (4) and (5) in the output paragraph

	below the stats	above the stats	tot
vect1	*n1b*	*n1a*	*n1*
vect2	*n2b*	*n2a*	*n2*
tot	*n1b+n2b=nb*	*n1a+n2a=na*	*n*

   and the hypergeometric prob. $P\{n1a\}$

7.
   a. limits of the confidence intervals for the difference between the statistics in 4 and 5 in the output paragraph [*l1,l2*] in 95% confidence level
   b. for each appropriate pair of the above stat., if it fell into its c.i.* or not

The programmer winced at these specifications and knew that he had his work cut out for him..

3.4 Refinement of Specifications using Abstract Data Types

In an attempt to refine the specifications, the programmer asked the client to explain what was going on. As the client was talking the programmer was hearing all sorts of statistical buzzwords. These buzzwords he had heard before; they sounded technically relevant, but their meaning was altogether not clear.

During this explanation, the client was completely confident that the programmer knew exactly what these terms mean and completely understood the specifications. When the programmer began to badger her with questions on the initial specifications, she was certain that the programmer was just acting as if he did not understand the specifications in order to get her to do his work. The client assumed that an experienced programmer would naturally understand statistics and could easily deduce what function was needed in the program much faster than she could ever understand the formalism for arriving at and writing the specifications. Thus, this client, as many others, impeded the process of arriving at the specifications by not cooperating as fully as possible in supplying information needed to produce the specifications. It took all the persistence at the programmer's disposal to wheedle the specifications out of the client.**

In any case, as the client talked, the programmer was hearing:

"...take an observation vector..."

*confidence interval

**Since completing this project, the statistician/client has become a computer scientist. In revising the final draft of this paper, the client looked at her own initial specifications for the first time in over two years and through the eyes of her new profession. She was astounded at these specifications — she could could not understand them any more and now sees that in fact the programmer did not understand them either.

"...truncate it..."

"...censor it..."

"...concatenate two observation vectors..."

"...T(time) of a vector at probability p..."

"...P(probability) of a vector at time t..."

"...the standard error of the vector at time *t* or probability p..."

etc.,

over and over again. The programmer began to see the abstract data type

observation vector

or simply, *vector*, where each observation is a real number and whose operations include

create

truncate

censor

concatenate

time

probability

delta sum

standard error

By nudging the client with "Are there any other operations on vectors that you're going to need for your program?", the programmer produced with the client's help a preliminary list of operations. The programmer and client then identified for each the list of parameter types and the return value type, if any.

Finally the programmer pressed the client for assurance that each of the operations is well understood by statisticians and either is trivially implemented (e.g., create, truncate, censor, etc.) or is defined by a well-known formula (e.g., time, probability, standard error, etc.). With this assurance finally extracted from the client, the programmer felt confident that the abstract data type and its operations were indeed implementable.

It was now time to begin refining the specifications into a useable document. The goal was to produce a description of the main program which used the abstract data type and its operations as if they were primitive. It was recognized that as the specifications were being refined,

1. additional required operations may be discovered, and

2. the parameter and return value types of existing operations may be changed.

When the specifications were complete any unneeded operations of the data type could be eliminated from the specification

Appendix 1 contains the final specifications including the *vector* abstract type. They are in the form that they were in at the completion of the project blemishes and all.

3.5 Final Input Specifications

To arrive at the final input specifications the programmer complained to the client that

1. spelled out identifiers would be clearer,

2. the program lacked generality, i.e., it could do only one run of the experiment and it does these experiments with both observation vectors the same size, with the same theta for censoring each vector, with the same loss rate in each case, and with the same truncation threshold in each case,

3. there was a magic constant, i.e., why was a 95% confidence level used as opposed to any other?

In addition the programmer demanded to know what were the bounds for each of the input values. The client's answers to these complaints and question led to the final input specifications that appear in Appendix 1.

3.6 Final Output Specifications

3.6.1 Strong Typing

As the output specifications were being refined, it became clear that it is basically a nest of loops whose purpose is to get certain functions printed out for each combination of values in their domains. Each loop is to step through the values of one parameter's domain. It seemed reasonable to nest the loops to permit as many possibly disjoint inner loops as possible to share outer loops consistent with the desired order of output.

Once the loop skeleton was set up, each operation application was put in its proper place in the skeleton. It was then checked that for each application of an operation, each of its parameters was either a global variable or the index for a loop nested about the application. In a number of cases, it was necessary to add a loop to provide in index which was used as the value of a parameter. Such additions, in turn, led to changing the nesting structure of the loops in order to better utilize sharing of outer loops.

It was also necessary to check for each use of an operation that the types of its actual parameters and the type value it was presumed to return agreed with those of other uses and with the parameter and return value types given with operation in the list of operations of the abstract type. In a number of cases, either within a version or across versions, there was not complete agreement. In other words, type checking failed. For example:

1. some of the applications of either of the *empirical standard error* operations had a size parameter of type **int**, and others did not. In the latter cases, the size is assumed to be that of the vector passed as another parameter and in the former cases, the size is independent of the vector. As assuming that size is independent of the vector is more general, the operations were finally specified to have a size parameter, and all applications of the size-less operation were modified by the addition of an appropriate size parameter.

2. some of the applications of either of *the theoretical standard error* operations had a theta parameter of type **real**, and others did not. It was observed that in any case, the theta parameter is not necessary, since one of the other parameters is calculated from theta, and the only use for the theta is in the calculation of this other parameter. Thus, the operation was finally defined with no theta parameter, and all applications with a theta as a parameter were modified accordingly.

One implication of some of these operation changes was to change the looping structure. For example, the first change above necessitated the introduction of loops to step through the different kinds of sizes including that of the vector determined by an outer loop.

Finally some operations on the original list were never used so they were eliminated from the list.

Observe that the analyses described above are completely syntax based and can be performed without having to know the semantics of the operations. These analyses include the following checks typically done by a compiler for a strongly typed programming language, namely that

1. each operation is used with the correct number and types of parameters;
2. each operation is used correctly as an expression of its return type;
3. each identifier used is declared as a variable, constant, or a loop index in some surrounding context;
4. each operation declared is used; and
5. each variable, constant, or loop index declared is used.

It is clear that the first three are more critical than the last two.

Apparently catching these syntactic inconsistencies was sufficient to find all inconsistencies, syntactic and semantic. In particular detecting missing variables, constants, loop indices and parameters by these syntactic means was sufficient to find all those that were missing.

3.6.2 Debugging Sense

In a few places, the programmer's debugging sense alerted him of some additional necessary output. This debugging sense said that a function value used as an actual parameter to another function should be printed out itself. For example in line 51 *time (vec,prob)* is used as the *t* parameter of the function *whether t lies within time empirical confidence limits*. Thus, *time (vec,prob)* should be printed out also within the same loop, as requested in line 19, even though it is also printed out earlier at the request of line 26 (It will not do to eliminate the request of line 26 as it is necessary to have a listing of all the *times* in one place.). As a result of the programmer's nagging, the client agreed to insert this extra printing and others like it.

The extra printing proved invaluable for the debugging as expected. Also it turned out that the client's advisor on several occasions requested some additional output which the client had not anticipated but which was there as a result of planning for debugging.

3.7 Programming, Testing, Debugging, Running, and Defense

The specifications were completed and jointly approved by the programmer, the client, and the client's advisor after about four weeks of arguing. At this point the client took over as the programmer. The program was then designed and coded by the client in FORTRAN to meet the specifications. The old programmer taught the client about structured programming and how to implement an abstract data type as a collection of subroutines and functions sharing access to a named COMMON area to which the main routine did not have access. The program ran in very few debugging runs. The bugs were easy to locate because of the extra debugging output. For the production runs, the WRITE statements for all output not required by the committee were commented out. The client finished in time and successfully defended the thesis before the committee.

3.8 Results

The results were amazing. Once the specifications were obtained and finalized (by sending them to be typed) they were not changed at all except to correct minor typographical errors.

1. There was none of the usual feedback on the specifications during the designing, coding, testing, debugging, and running of the program. In particular, *no* inconsistencies were uncovered, and *no* new functions were found necessary in any of the later stages of the program's lifecycle.
2. Even during the acceptance phase, i.e., the thesis defense, *no* new functions were discovered.

This is the first time that either author has observed this phenomenon, particularly the first. The first author is well aware of how rare such an occurrence is. What made the occurrence even more amazing is the fact that the programmer and the client came from different disciplines and clearly had the potential for the usual misunderstandings between programmer and client. Indeed to date, the first author cannot talk authoritatively about the program without double-checking with the second author.

That no new functions were discovered during acceptance, i.e., during the thesis defense, was due to a combination of sheer luck and the shortness of the acceptance phase. That is the second author's advisor was leaving the country very shortly thereafter, and the entire committee knew it. Still the functionality of the resulting program was quite complete.

There are, however, good reasons to believe that the lack of feedback on to the specifications during design, implementation, and testing was no accident. There are resons to believe that the use of strong typing, abstract data typing, and programmer persistence is sufficient to yield consistent, change-resistant specifications. There are studies, e.g., [Egg81], pointing to the value of the redundancy offered by strong typing in catching errors early. There are indications, e.g., in [HPU81], of the value of abstract data typing and information hiding in producing designs that withstand modifications as implementation details are filled in. There are no modifications to the design because all of the changes that are made are to the information that is hidden anyway. Finally, the value of nudging in seeing a job through to its proper conclusion is clear.

4. Specification Languages

This experience also suggests certain properties that a specification language should have. First, it should have the ability to exhibit the modules of the specified system, particularly the data abstraction modules. Secondly, it should be strongly typed, that is, it should be possible to specify the types of all data objects including parameters so that type and interface consistency can be checked at the *time the specification is being written* (and not later as the program is written). Thirdly, it should have a processor which does all the type and interface consistency checking possible.

This last requirement is essential. Without machine enforcement of type and interface restrictions, they do not get obeyed. Even if there is a desire to obey them fastidiously, mistakes can be made. Indeed, later, as an opportunity arose to express the specifications in a formal language and to have them checked by a processor, errors were found (See below.). None were serious. In all cases what was meant was clear, and thus these errors did not negate the value of the specifications. Had there been a processor for the specification language, these errors would have been caught, and the programmer could have been confident that there would be no type and interface errors. The point is, writing specifications is error-prone as is. If any part of a specification can be automatically checked, it should be in order to eliminate at least all the avoidable, stupid errors.*

Subsequent to writing the first draft of this paper, the programming language Ada appeared on the scene. Ada allows construction of data abstractions with its *package* feature, and it is strongly typed. Its processors are required to do type and interface checking at compile time. In addition, Ada has the nice property that a complete program does not have to be presented to the compiler in order for it to do type and interface checking. A module can be completely type and interface checked in the presence of only the *specification parts* of the subprograms and packages it makes use of. The specification part of a subprogram is basically its header giving its name, parameter types, if any, and return value type, if any. The specification part of a package is basically the exported types, constants, etc. and the headers of the exported subprograms of the package. As a consequence of this property of Ada, there have been proposals for using Ada as a program design language [Wau80]. A heavily commented program skeleton consisting mainly of specification parts only is submitted to an Ada compiler which then does as much type and interface checking as possible.

The first author tried expressing the specification in a sort of an Ada program design language. The complete result is given in Appendix 2 of [BB82]. The *vector* abstract type is given as a syntactically improper Ada package specification part for the package *observation vector.*

```
with TEXT_IO; use TEXT_IO;
package OBSERVATION_VECTOR is
```

* A *stupid error* is any error that can algorithmically avoided.

```
    type VECTOR is private;
    function create_observations(size:INTEGER;theta:FLOAT)return VECTOR;
    function create kept_observations(v:VECTOR;loss_rate:FLOAT;
          name:STRING)return VECTOR;
    function create truncated observations(v:VECTOR;time:FLOAT;
          name:STRING)return VECTOR;
    function concat(v1,v2:VECTOR)return VECTOR;
    function size(v:VECTOR)return INTEGER;
    function time(v:VECTOR;prob:FLOAT)return FLOAT;
    function probability(v:VECTOR;time:FLOAT)return FLOAT;
    function delta sum(v:VECTOR)return INTEGER;
    function time theoretical std error(size:INTEGER;prob:FLOAT)
          return FLOAT;
    function time empirical_std error(size:INTEGER;v:VECTOR;
          prob:FLOAT)return FLOAT;
    function prob theoretical std error(size:INTEGER;time:FLOAT)
          return FLOAT;
    function prob empirical std error(size:INTEGER;v:VECTOR;
          time:FLOAT)return FLOAT;
    function above(v:VECTOR;time:FLOAT)return INTEGER;
    function below(v:VECTOR;time:FLOAT)return INTEGER;
    procedure print name(v:VECTOR);
    function difference std error(val1,val2:FLOAT)return FLOAT;
    function theta(v:VECTOR)return FLOAT;
end OBSERVATION VECTOR;
```

The package specification part is improper because it has no private part, but it was nevertheless accepted by the compiler. The main program is given in a bastardized Ada written in a manner to fool the processor into doing the required checking. That is, natural language material is buried inside comments, and any text containing a use of a subprogram of the package is exhibited in proper Ada syntax. Where possible, Ada notation corresponding to the original notation is used. For example, Ada declarations, for loops, *get*, and *put* are used. In some cases, preserving correspondence leads to improper Ada, e.g.,

```
for each time ∈ {½,1,2} do
     .
     .
od;
```

is expressed as

```
for t in {1/2,1,2} loop
     .
     .
end loop;
```

which would be Ada except for the set notation and the fact that *1/2,1,2* do not form a proper enumeration. This Ada fragment is made a comment and in its place is put block containing a loop:

```
declare t:FLOAT;
        time set:constant array(1..3)of FLOAT:=
                        (1.0/2.0,1.0,2.0);
begin
        for i in time set'RANGE loop
                t:=time set(i);
                     .
                     .
```

```
          end loop;
     end; .
```

This substitute gets the compiler to do all the checking it would have to do with the improper statement, i.e., that 1/2, 1, and 2 are proper to assign to *t* and that in the scope of the loop, *t* is used properly for its type. The generation of the substitute statements is fairly straightforward and can be automated.

Valid Ada value-returning expressions sitting in the middle of a natural language sentence have to be brought out of the comment containing the sentence so that they could be type and interface checked. However, the Ada compiler that was used [NYU81] complains (and for that matter any should) when such expressions are left sitting in the position of statements, and refuses to do any checking whatsoever. Thus internal procedures with the natural language sentences as their names were invented. These procedures have formal parameters for each of the expressions embedded in their sentences. An application of these procedures to these expressions is acceptable to the compiler and gets thoroughly checked. Occasionally it was convenient to introduce a local variable to explicitly hold the value designated by a pronoun or a pronoun-like phrase. As an example, the body of the loop in lines 48-54 is translated as

```
declare p:FLOAT;
        prob set:constant array(1..3)of FLOAT:=
                        (0.25,0.5,0.75);
begin
        for i in prob set'RANGE loop
              p:=prob set(i);
           put(time(vec,p));
           conf limits:=time empirical confidence limits(
                 theta(vec),p,
                 time empirical std error(
                    n,vec,p),
                 z value(k));
                 put(conf limits);
           put(whether(time(vec,p),lies within=>conf limits));
           conf limits:=time theoretical confidence limits(
                 theta(vec),p,
                 time theoretical std error(
                    n,p),
                 z value(k));
                 put(conf limits);
           put(whether(time(vec,p),lies within=>conf limits));
        end loop;
end; .
```

Note that the sentence "whether ... lies within ..." was converted into a function *whether* specified with

```
function whether(v:FLOAT,lies_within:LIMITS)return BOOLEAN; .
```

Thus, the Ada specification has more details than the original specification and perhaps more than is desirable in a specification.

The specification was thus type and interface checked. Some errors in the original specifications were found. Many are minor punctuation and spelling errors. Some of these are typographical and transcription errors.

There were four interface errors discovered, all of exactly the same kind. The definition of the *vector* abstract type shows that *create kept observations* and *create truncated observations* each require a third parameter, a string giving the name of the vector to be created. The four calls to these two routines, which are in lines 14, 15, 16, and 17 of Appendix 1, each has only two actual parameters. In particular, the name string is missing. This error was found when the Ada compiler complained about the missing parameters in the Ada versions of these calls. Interestingly, the FORTRAN code for these calls passes all three parameters correctly in each case. Thus, only the specifications have the interface error. This error completely slipped by the two authors, the adviser, and the committee despite careful scrutiny. It even slipped by in proof readings of earlier versions of this paper.

The net effect of this error is not severe, as the code does not make this error. However, the nature of this error, an interface error, is quite serious. Therefore, the importance of having a processor to find errors is borne out. Had there been a processor for the original specification language, none of the errors, minor or not, would have remained in the specifications.

The form of the Ada version of the specifications leaves a bit to be desired. Too many tricks had to be played to get the compiler to do all the desired checking. These tricks included having to invent procedures for sentences in order to force type checking of embedded significant phrases. A proper program design language might be better suited. Such a program design language would have to accept a mixture of natural language sentences and programming language statements. Its processor would have to be able to distinguish between the two so that it could do the checking that it is supposed to even if the call to a package subprogram is buried in the middle of an natural language sentence. It would also have to not complain when an expression buried in a natural language sentence seems not to be used properly for its type because the sentence of course does not define what type value it requires from the contained expression.

A better candidate for expressing the specifications and getting the desired type and interface checking done is the program design language SDP [Lin80, LYB81]. It permits natural language sentences to be designated as the header of a subprogram and as the name of a data abstraction. In the defining occurrences of these sentences, any of its words may be designated as formal parameter type names. A sentence which agrees word-for-word with such a defining sentence in all but the formal parameter words is considered an application (call) of the sentence. A word of the application sentence which is in the position of a formal parameter is checked to be declared with the formal parameter word as its type, if it is declared to be of any type at all.

The specification was expressed in approximately the language of SDP. Keywords show up in bold face, and formal and actual parameters get underlined or left-sidelined. Only the formal parameters are marked specially on input. The actuals get marked as a result of the processor's recognizing the containing sentence as an application of a defining sentence. The complete specification is found in Appendix 3 of [BB82]. Here is shown the SDP version of only the package specification and the loop discussed above:

```
cluster observation_vector

        op create observation vector of length size with distribution theta
        op create kept observation vector from observation_vector according to
                        loss rate lr named name
        op create truncated observation vector from observation_vector
                        truncated at time t named name
        op concatenate observation_vector and observation_vector
        op size(observation_vector )
        op time of observation_vector at probability p
        op probability of observation_vector at time t
        op delta sum(observation_vector )
        op time theoretical std error of any vector of length size at
                        probability p
        op time empirical std error of observation_vector of length
                        size at probability p
        op prob theoretical std error of any vector of length size at
                        time t
        op prob empirical std error of observation_vector of length
                        size at time t
        op number of entries in observation_vector above time t
        op number of entries in observation_vector below time t
        op print name of observation_vector
op difference std error of value1 and value2
op theta of observation_vector
```

and

```
declare p as real
do for p in {0.25,0.5,0.75}

        time of vec at probability p
        time empirical confidence limits
                theta of vec
                p
                time empirical std error of vec
                        of length n at probability p
                z_value(k)
        whether
                time of vec at probability p
                lies within time empirical
                confidence limits
        time theoretical confidence limits
                theta of vec
                p
                time theoretical std error of any vector
                        of length n at probability p
                z_value(k)
        whether
                time of vec at probability p
                lies within time theoretical
                confidence limits

od
```

The resulting specification is closer to the original specification than the Ada version. It is also uses a language a bit more general than the current version of SDP; it uses full sentences as actual parameters in other sentences. These can be spotted as sentences (which themselves may have underlined actual parameters) which are left-sidelined and are embedded inside other sentences. No processor exists yet for this extended SDP, but it is clear that the requisite pattern recognition and type and interface checking can be done by an augmented SDP processor.

5. Conclusions and Future Research

This report has described an experience of the authors in arriving at specifications that withstood the usual modification-causing feedback from the program's later life cycles despite the clear potential for misunderstanding. On the basis of this experience, a methodology for arriving at good, complete, consistent, and change-resistant specifications is proposed. This methodology, which is useable even when the programmer and the client speak different jargons, makes use of the ideas of abstract data typing, strong typing, and programmer persistence. In addition, recommendations were given concerning specification languages and their processors.

It is necessary to examine the general applicability of the proposed methodology. Did the methodology succeed because of itself, because of the personalities of the programmer and the client, because of the particular nature of the problem, etc.? If the methodology works only for certain kinds of problems, what are they? It may also be useful to conduct controlled experiments over a wide variety of problems with large numbers of programmer-client pairs.

6. Acknowledgements The authors thank an anonymous referee for helpful comments.

7. Bibliography

[ADA81] "Reference Manual for the Ada Programming Language", U. S. Department of Defense, MIL-STD-1815 (December 1981).

[BB81] Berry, D.M. "The Programmer-Client Interaction in Arriving at Program Specifications: Guidelines and Linguistic Requirements", Computer Science Department, UCLA (1982).

[Bry79] Berry, O. "Comparison Between Two Life Span Distributions Based on Small Samples with Censored Data", M.A. Thesis, Dept. of Statistics, Tel Aviv University (1979).

[DB77] Dixon, W.J. and M.B. Brown (Eds.), *BMDP-77 Biomedical Computer Programs P-series*, Berkeley, University of California Press (1979).

[DOD78] "Requirements for High Order Computer Programming Languages, STEELMAN", Department of Defense (1978).

[Egg81] Eggert, P.R. "Detecting Software Errors before Execution", Ph.D. Thesis, Computer Science Dept., UCLA (1981).

[Flo75] Flon, L., "Program Design with Abstract Data Types", Dept. of Computer Science, Carnegie- Mellon University (1975).

[Gre64] Greenberg, D., *How to be a Jewish Mother*, Los Angeles, Price/Stern/Sloan (1964).

[HPU81] Hester, D.L., D. Parnas, and D.F. Utter, "Using Documentation as a Software Design Medium", *Bell Systems Technical Journal*, 60:8, 1941-1977 (October 1981).

[Lin80] Linden, N.M., "Software Development Processor User Reference Manual", Mayda Software Engineering, P.O.B. 1389, Rehovot, Israel (1980).

[LYB81] Linden, N.M., M. Yavne, and D.M. Berry, "Parameterization and Abstract Data Types in a Program Design Language: The Design of Software Development Processor", *Primera Conferencia Internacional en Ciencias de la Computacion*, Santiago, Chile (August 1981).

[LZ74] Liskov, B.H., and S.N. Zilles, "Programming with Abstract Data Types", *SIGPLAN Notices* 9:5 (1974)

[Mye78] Myers, G.H., *Composite/Structured Design*, New York, van Nostrand Reinhold (1978).

[NYU81] "The NYU Ada/Ed System, An Overview", Courant Institute, New York University (July 1981).

[Par72] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules", *CACM* 15:12 (1972)

[vWn75] van Wijngaarden, A. *et al* (Eds.), "Revised Report on the Algorithmic Language ALGOL 68", *Acta Informatica* 5 (1975).

[Wau80] Waugh, D.W., "Ada as a Design Language", *IBM Software Engineering Exchange* 3:1 (October 1980).

Appendix 1

Specification of what is input and output

INPUT

```
int  size 1, size 2; ¢ sample sizes  1 ≤  ≤ 50 ¢
real  theta 1, theta 2; ¢ parameter of exponential dist > 0 ¢
int  no of loss rates ¢ 1 ≤  ≤ 5 ¢
[0: no of loss rates] loss rate; ¢ 0 ≤ LR[i] ≤ 1 ∧ LR[0] = 0 ∧ all others are
read in ¢
real  threshold; ¢ truncation threshold > 0 ¢
[1:5] real confidence level; ¢ 0 ≤ CL[i] ≤ 1 ¢
int repetitions; ¢ no of times do the experiment ¢
[1:5] real z-value ¢ 0 ≤ z,v[i] ≤ 4 ¢
```

OUTPUT

(using vector type as defined on a separate page)

```
for  i  from  1  to repetitions do
   observs 1 = create observations (size 1, theta 1);¢ sorted ¢
   observs 2 = create observations (size 2, theta 2);¢ sorted ¢
   loss rate [0] = 0;
   for j from 0 to no of loss rates do
      kept observs 1 = create kept observations (observs 1, loss rate [j]);
      kept observs 2 = create kept observations (observs 2, loss rate [j]);
      trunc observs 1 = create truncated observations (kept observs 1,
      threshold);
      trunc observs 2 = create truncated observations (kept observs 2,
      threshold);
      concat kept obs = concat (kept observs 1, kept observs 2)
      concat trunc obs = concat (trunc observs 1, trunc observs 2)
      for each vector ∈ {kept observs 1, kept observs 2, concat kept obs,
                          trunc observs 1, trunc observs 2, concat trunc obs}
                 do
         for each prob ∈ {.25,.5,.75} do
            time (vector, prob)
         od;
         for each time ∈ {½,1,2} do
            prob (vector, time)
         od
      od;
      for each (vec 1, vec 2) ∈ {(kept observs 1, kept observs 2),
                                  (trunc observs 1, trunc observs 2)} do
         for each prob ∈ {.25,.5,.75} do
            table and hypergeometric probability (
                  above (vec 1, time (concat (vec 1, vec 2), prob)),
                  below (vec 1, time (concat (vec 1. vec 2), prob)),
                  above (vec 2, time (concat (vec 1, vec 2), prob)),
                  below (vec 2, time (concat (vec 1, vec 2), prob)),
                  "for probability=",prob)
         od;
         for each time ∈ {½,1,2} do
            table and hypergeometric probability (
                  above (vec 1, time), below (vec 1, time),
                  above (vec 2, time), below (vec 2, time),
                  "for time=",time)
         od
      od ;
      for each vec  ∈ {kept observs 1, kept observs 2, trunc observs 1,
                  trunc observs 2}
      do
         for each N ∈ {size (vec), delta sum (vec),
                  $\frac{\text{size (vec) + delta sum (vec)}}{2}$} do
            for k from 1 to 5 ¢ = no of confidence levels ¢ do
               for each prob ∈ {.25,.50,.75} do
                  time (vec, prob);
                  time empirical confidence limits (theta (vec), prob,
                       time empirical std error (N, vec, prob),Z-value [k]);
                  whether time (vec, prob) lies within time empirical
                       confidence limits;
```

```
                time theoretical confidence limits (theta (vec), prob,
                    time theoretical std error (N, prob),
                    Z-value [k]);
                whether time (vec, prob) lies within time theoretical
                    confidence limits
            od;
          for each time ∈ {½,1,2} do
                prob (vec, time);
                prob empirical confidence limits (theta (vec), time,
                    prob empirical std error(N,vec,time), Z-value [k]);
                whether prob (vec, time) lies within prob empirical
                    confidence limits
                prob theoretical confidence limits (theta (vec), time,
                    prob theoretical std error ( N, time),
                    Z-value [k]);
                whether prob (vec, time) lies within prob theoretical
                    confidence limits
            od
          od
        od
      od;
      for each (vec 1, vec 2) ∈{(kept observs 1, kept observs 2),
                    (trunc observs 1, trunc observs 2)} do
        for each ($N_1$,$N_2$) ∈ {(size (vec 1), size (vec 2)),(delta sum(vec 1),
                  delta sum(vec 2)), ($\frac{[size (vec 1) + delta sum (vec 1)]}{2}$,
                      $\frac{[size (vec 2) + delta sum (vec 2)]}{2}$ )}
          for k from 1 to 5 ¢ = no of confidence levels ¢ do
            for each prob ∈ {.25,.50,.75} do
                time difference = time (vec 1, prob) - time (vec 2, prob);
                time difference empirical confidence limits (theta (vec 1)
                    theta (vec 2), prob, difference standard error (time
                    empirical std error ($N_1$, vec 1, prob), time empirical
                    std error ($N_2$, vec 2, prob)), Z-value [k]);
                whether time difference lies within its empirical
                    confidence limits;
                time difference theoretical confidence limits (theta(
                    vec 1), theta (vec 2), prob, difference standard
                    error (time theoretical std error ($N_1$,
                    prob), time theoretical std error ($N_2$,
                    prob)), Z-value [k]);
                whether time difference lies within its theoretical
                    confidence limits
            od;
            for each time ∈ {½,1,2} do
                prob difference = prob (vec 1, time) - prob (vec 2, time)
                prob difference empirical confidence limits (theta (vec 1),
                    theta (vec 2), time, difference standard error (prob
                    empirical std error($N_1$, vec 1, time), prob empirical
                    std error ($N_2$, vec 2, time)), Z-value[k]);
                whether prob difference lies within its empirical
                    confidence limits;
                prob difference theoretical confidence limits (theta (
                    (vec 1), theta (vec 2), time, difference standard
                    error (prob theoretical std error (
                    $N_1$, time), prob theoretical std error (
                    $N_2$, time)), Z-value [k]);
                whether prob difference lies within its theoretical
                    confidence limits
            od
          od
        od
      od
    od
od;
for each dist ∈ {"empirical dist", "theoretical dist"} do
  for each case ∈ {"kept observs", "trunc observs"}
    for each kind of N ∈ {"obtained from size (vec)", "obtained from delta
          sum (vec)", "obtained from $\frac{size (vec) + delta sum (vec)}{2}$"} do
      for each theta ∈ {theta 1, theta 2}
        for each prob ∈ {.25,.5,.75} do
```

```
92              print time summary table (prob, theta, dist, kind of N,
                    no of repetitions, case)
93           od
94           for each time ∈ {½,1,2} do
95              print prob summary table (time, theta, dist, kind of N,
                    no of repetitions, case)
96           od
97        od
98        for each prob ∈ {.25,.5,.75} do
99           print time difference summary table (prob, theta 1, theta 2,
                 dist, kind of N, no of repetitions, case)
100       od
101       for each time ∈ {½,1,2} do
102          print prob difference summary table (time, theta 1, theta 2,
                 dist, kind of N, no of repetitions, case)
102       od
103    od
104  od
105 od
106 where:          time summary table (prob, theta, kind of N, no of
                         repetitions, case)=
                    "TIME (OBS, "prob")"
                    theta
                    dist ¢ empirical , theoretical ¢
                    kind of N
                    no of repetitions
                    case ¢ kept observs , trunc observs ¢
```

confidence level / loss rate	[j]
[i]	percentage of the no of repetitions in which "TIME(OBS, "prob")" lies within its confidence interval limits obtained for loss rate [i] and confidence level [j] using its std error computed according to the dist (empirical or theoretical)

```
107 where:          prob summary table (times theta, dist, kind of N,
                    no of repetitions, case)=
                        "PROB (OBS, "time")"
                        dist ¢ empirical, theoretical ¢
                        kind of N
                        no of repetitions
                        case ¢ kept observs, trunc observs ¢
```

confidence level / loss rate	[j]
[i]	percentage of no of repetitions in which "PROB (OBS, "time")" lies within its confidence interval limits obtained for loss rate [i] and confidence level [j] using its std error computed according to the dist (empirical or theoretical)

```
108 where:          time difference summary table (prob, theta 1, theta 2,
                    dist, kind of N, no of repetitions, case)=
                        "TIME (OBS 1, "prob")-TIME (OBS 2, "prob")"
                                    theta 1  theta 2
                        dist ¢ empirical, theoretical ¢
                        kind of N
                        no of repetitions
                        case ¢ kept observs, trunc observs ¢
```

confidence level / loss rate	[j]
[i]	percentage of no of repetitions in which "TIME (OBS 1, "prob")-TIME (OBS 2, "prob")" lies within its confidence interval limits obtained for loss rate [i] and confidence level [j] using its std error computed according to the dist (empirical or theoretical)

109 **where:** prob difference summary table (time, theta 1, theta 2, dist, kind of N, no of repetitions, case)=

```
    "PROB (OBS 1, "time")-PROB (OBS 2, "time")"
    theta 1, theta 2
    dist ¢ empirical, theoretical ¢
    kind of N
    no of repetitions
    case ¢ kept observs, trunc observs ¢
```

loss rate \ confidence level	[j]
[i]	percentage of no of repetitions in which "PROB (OBS 1, "time")-PROB (OBS 2, "time")" lies within its confidence interval limits obtained from loss rate [i] and confidence level [j] using its std error computed according to the dist (empirical and theoretical)

110 **where:**

table and hypergeometric probability (int above 1, below 1, above 2, below 2, character label, real val)=

label val:

	Below	Above	Total
kept observs 1	below 1	above 1	above 1 + below 1
kept observs 2	below 2	above 2	above 2 + below 2
Total	below 1 + below 2	above 1 + above 2	(above 1 + above 2 +below 1 + below 2)

$$\text{hypergeometric probability} = \frac{\binom{\text{total 1}}{\text{above 1}}\binom{\text{total 2}}{\text{above 2}}}{\binom{\text{over all total}}{\text{total above}}}$$

Vector Abstract Data Type

```
type vector
   create observations (int size, real theta) vector
   create kept observations (vector v, real loss rate, string name) vector
   create truncated observations (vector v, real time, string name) vector
   concat (vector v1, V2, string name) vector
   size (vector v) int
   time (vector v, real prob) real
   probability (vector v, real time) real
   delta sum (vector v) integer
   time theoretical std error (int size, real prob) real
   time empirical std error (int size, vector v, real prob) real
   prob theoretical std error (int size, real time) real
   prob empirical std error (int size, vector v, real time) real
   below (vector v, real time) int
   above (vector v, real time) int
   print name (vector v)
   difference std error (real val 1, val 2) real
   theta (vector v) real
end vector
```