UNC Report No.
TR 77-012

December 1977

USING TRACES TO WRITE ABSTRACT SPECIFICATIONS FOR SOFTWARE MODULES

Wolfram Bartussek and David L. Parnas

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27514 U.S.A.

USING TRACES TO WRITE ABSTRACT SPECIFICATIONS FOR SOFTWARE MODULES

Wolfram Bartussek
and
David L. Parnas
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27514 U.S.A.

ABSTRACT

A specification for a software module is a statement of the requirements that the final programs must meet. In this paper we concentrate on that portion of the specification that describes the interface between the module being specified and other programs (or persons) that will interact with that module. Because of the complexity of software products, it is advantageous to be able to evaluate the design of this interface without reference to any possible implementations. The first sections of this paper present a new approach to the writing of black box specifications, illustrate it on a number of small examples, and discuss checking the completeness of a specification. Section VIII is a case history of a module design. Although the module is a simple one, the early specifications (written using an earlier notation) contained design flaws that were not detected in spite of the involvement of several persons in a series of discussions about the module. These errors are easily recognized using the method introduced in this paper.

I. THE ROLE OF SPECIFICATIONS IN SOFTWARE DESIGN

We are concerned with the building of software products that are so large that we cannot manage the task unless we reduce it to a series of small tasks. We further assume that each of the subtasks (which we call modules) will focus on one portion of the design and hide the details of that aspect of the design from the rest of the system. This has become known as the "information hiding principle," encapsulation, data abstraction, etc. [1,2,3]. The design process will only go smoothly if the inter-module interfaces are precisely defined. Ideally, the interface description states only the requirements that the component must satisfy and does not suggest any other restrictions on the implementation. We term such a description of the requirements a <u>specification</u> [19]. We also note that any software product is but a module in a still larger system; its requirements should be specified as precisely as each of its components.

For a trouble-free development process it is also necessary that one be able to verify the reasonableness of decisions before proceeding to make further decisions. If we reverse one of our decisions later (or find that it was inadequately described), we may have to discard all work done subsequent to that decision . If we have written a formal specification for a module, we should be able to verify that the specification has such basic properties as consistency and completeness. Techniques for doing that will be discussed later in this paper.

II. WHAT ARE SPECIFICATIONS?

A fair amount of confusion has been caused by the fact that the word "specification" is used with two distinct meanings in the computer literature. The dictionary definitions of the word "specification" cover any communication which provides additional information about the object being described - any communication that makes the description of the object more specific. In engineering usage, the word has a narrower meaning. A specification is a precise statement of the requirements that a product must satisfy. A description of the number of ones in the binary representation of a computer program is a specification in the general sense but it is rarely a specification in the engineering sense.

In the remainder of this paper we will use the engineering sense of "specification."

III. BRIEF HISTORY OF WORK ON SOFTWARE SPECIFICATIONS

We distinguish two classes of specifications for software, which we shall denote as P/P (Precondition-Postcondition) and DA (Data Abstract). P/P specification techniques are based on the pioneering work of Floyd [4] and subsequent work by Hoare [5], Dijkstra [6], and others. P/P techniques describe the effect of a program in terms of predicates that describe acceptable states of data structures. The Precondition is a predicate that describes the states in which the program may be stated. The Postcondition describes the states after program termination. Dijkstra's predicate transformers replace both of these predicates by a rule for transforming a postcondition into a precondition [6,7]. P/P specifications describe the change of state that the program must effect, but not how to effect it. Usually, the effect of each individual program is described separately and in terms of the data structure accessed by the program.

In DA specifications the specification of a module does not refer to the data structure used within a module.

That data structure is not part of the requirement; it is part of the solution. It does not belong in a statement of requirements because it is an implementation decision. Early work on specifications that "hide" implementation data structures was done by Parnas [8]; more recent work by Guttag [9,10] put a sounder mathematical basis behind the work and suggested some notational improvements.

The DA specification work is motivated by a desire to give a "black-box" description of a software module. The user is told only of a set of access programs. Some of these (here termed V-functions) return values that give information about the data stored within the module. Others (here termed O-functions) change the internal data. In most cases, the execution of an O-function will eventually cause a change in the value of a V-function. The effects of the call of the O-function may not be visible in terms of V-function values until some other C-functions have been executed.

Parnas's early work was done on an ad hoc basis. The notation was developed to meet the needs of specific examples [8]. The early examples had the property that the effects of O-functions were immediately visible and could be described in terms of the new values of the V-functions. Only in later examples did Parnas and Handzel [20] seek to extend these techniques to cases where there were delayed effects.

The problem of delayed effects led Price and Parnas [10,11,12] to include "hidden" functions in their specifications. The "hidden" functions are not available outside the black box. They need not be implemented; their purpose is purely descriptive. The effects of O-functions are described in terms of the values of the hidden functions. These hidden functions are still in use at SRI [13] and elsewhere.

In spite of all disclaimers, the hidden functions do suggest data structures and possible implementations of the program. Liskov [14] and others have suggested writing specifications simply by giving possible implementations - i.e., by giving a program whose behavior would be acceptable and asking that the programs produced be "equivalent."

The equivalent program approach and the hidden functions disturb us. They violate the basic motivation for DA specifications by providing information that is not a requirement. Some of the properties of this hypothetical implementation may not be required of the actual program. "One must be very careful not to read too much into such specifications" [14].

Guttag's method does not rely on hidden functions to describe delayed effects. His papers [9,10] describe a systematic way of writing the specification. However, there were cases that he could not handle without the introduction of hidden functions. One of those examples, the stack with overflow, will be used later in this paper [15].

In this paper, we propose yet another approach. It allows the specification of modules with delayed or hidden effects without any reference to internal data structures. The only statements made are about the effects of calls on user accessible O-functions on user accessible V-functions.

IV. WHEN IS A D/A SPECIFICATION COMPLETE?

For simplicity, we assume that our modules are always created in the same initial state and could be returned to that state (reinitialized). We further assume that for each access program (O-function or V-function) there is an applicability condition. If this condition holds, the program may be called. In states where the condition does not hold, the module will "trap" or refuse to return through the normal exit [16]. Values of V-functions after a trap occurs will not be discussed in this paper.

A trace of a module is a description of a sequence of calls on the functions starting with the module in the initial state. A trace is termed a legal trace if calling the functions in the sequence specified in the trace with the arguments given in the trace when the module is in its initial state will not result in a trap. A specification completely determines the externally visible behavior of a module if for every legal trace ending with a call of a V-function, the value returned by that V-function can be derived from the specification. We term such a specification complete. A specification is consistent if only one value can be derived.

There are situations in which one may want a specification that is not complete in the above sense. In this paper, however, we will concern ourselves with the problem of recognizing complete and consistent specifications.

V. A FORMAL NOTATION FOR SPECIFICATION BASED ON TRACES

A specification will consist of two main parts. The first part, which we call syntax, gives the names of all of the access programs, and the type of each of the parameters. For O-functions we will indicate that it changes an object of the type being specified. For V-functions we will give the type of value that it delivers. This information is necessary for recognizing whether a program using the functions could be compiled by a typical compiler. The notation used is that used by Guttag. [9,10]

The second part of the specification will be called the semantics. It consists of three types of assertions.

(1) Assertions about trace legality. These assertions identify a subset of the set of legal traces, that is a set of traces such that calling the functions as described in the trace (starting with a module in its initial state) will not result in traps . Additional legal traces may be implied by the equivalence assertions (see below). Any traces that cannot be shown to be legal using these assertions will be considered illegal traces.

(2) Assertions about the equivalence of traces. These assertions specify an equivalence relation on traces, such that (1) equivalent traces have the same legality (either both are legal or both are not legal) and (2) that they have the same externally visible effect on the module or data item. These assertions of equivalence will often enable us to extend the class of traces known to be legal. Equivalence is usually weaker than equality. Two traces are equal if they are identical in every respect (the same sequence of function calls with the same parameters).

(3) Assertions about the values returned by V-functions at the end of traces. These statements describe the values delivered by V-functions for a subset of the set of legal traces. The traces discussed directly in this section of a specification are called normal form traces. Using the equivalence statements, one can derive the values of V-functions at the end of other traces by finding an equivalent normal form trace.+

The three classes of assertions form a specification or statement of requirements. An implementation will be considered correct if and only if the assertions are true of it. Any property that one can deduce from the assertions must be a property of any correct implementation.

A program that uses the module in such a way that the program's correctness depends only on properties of the module that can be deduced from the specification's assertions will be able to use any correct implementation of the module.

+ In our examples, we have assumed that equality is defined for values of the types returned by the V-functions. In the unlikely event that we have no equality operator, V-function values would have to be described in terms of the operators that are available.

NOTATION

(1) Notation for describing the syntax (taken from Guttag).

<Function Name>: <type of parameter>X,...X<type of parameter> -> <type of result>

If the module maintains only one data item, that parameter need not be explicitly named in each function call.

(2) Notation for describing traces.

A trace will be represented as a string from the language described by the following syntax. The parsing of a trace into component subtraces is deliberately ambiguous. The trace denotes execution of the functions named in a left to right sequence.

```
<subtrace> ::=      ⊔ |<syntactically.correct.function.call>|
                    <subtrace>.<syntatically.correct.function.
                    call>

<trace> ::=         ⊔ |<subtrace>[.<subtrace>]*
```

[<T>]* denotes any number of occurrences of <T>.

"⊔" denotes an empty trace. Note that the symbol "⊔" never occurs in a trace.

We will frequently use the following shorthand notation.

Let p_i, $m \leq i \leq n$, be a list of actual parameters and $X(p_i)$ a syntactically correct function call. Then $X_M^N(p_i)$ denotes the same as

$$X(p_M).X(p_{M+1}). \cdots .X(p_{N-1}).X(p_N)$$

If the list of parameters is empty, then X_M^N is simply X.X. .. .X with n-m+1 repetitions of X. If m>n, then X_M^N denotes the empty trace. For $N \geq 1$ we write $X_1^N(p_i)$ as $X^N(p_i)$.

It is always assumed that a function call correctly adheres to the rules of the syntax section.

(3) Describing legality of sequences.

We introduce the predicate $\mathcal{E}(T)$ where T is a trace. $\mathcal{E}(T)$ is true if T is a legal trace. The appearance of the assertion $\mathcal{E}(T)$ in a specification is a requirement that calling the functions as described in T will not result in a trap.

Assuming that the module will not "trap" if it is not used, we *always* assume $\mathcal{E}(\sqcup)$ = *true*. (The empty trace is always legal.) It follows from our discussion of traces that if T is a trace and S is a subtrace

$$\mathcal{E}(T.S) \Rightarrow \mathcal{E}(T).$$

In other words, the prefix of any legal trace is a legal trace.

(4) Describing the values of V-functions at the end of traces.

If T is a legal trace, X is a syntactically correct call on a V-function, and $\mathcal{E}(T.X)$ is TRUE, then $V(T.X)$ describes the value delivered by X when called after an execution of T.

(5) Describing equivalence of two traces.

If T_1 and T_2 are traces then asserting that $T_1 \equiv T_2$ is an assertion that:

for any subtrace S (including the empty subtrace),
$$\mathcal{E}(T_1.S) = \mathcal{E}(T_2.S)$$

for any subtrace S (including the empty subtrace)
and V-function X,
$$\mathcal{E}(T_1.S.X) \Rightarrow V(T_1.S.X) = V(T_2.S.X)$$

Then ≡ is an equivalence relation. Note that the equivalence of two traces does not imply that they are the same in every respect, only in those respects specified above. For example, one may *not* conclude that two equivalent traces have the same length or that the prefixes of equivalent traces are equivalent. Note too that the above does not define a particular equivalence relation; that is done in each specification.

In the following specifications we have omitted universal quantifiers for variables representing traces (T) and integers a.

VI. SOME SIMPLE EXAMPLES (To be explained and discussed in Section VII.)

Example 1. A Stack for Integer Values

Syntax:

```
     PUSH:   <integer> x <stack> -> <stack>
      POP:               <stack> -> <stack>
      TOP:               <stack> -> <integer>
    DEPTH:               <stack> -> <integer>
```

Semantics:

A. Legality:

(1) $\mathcal{E}(T) \Rightarrow \mathcal{E}(T.PUSH(a))$
(2) $\mathcal{E}(T.TOP) = \mathcal{E}(T.POP)$

B. Equivalences:

(3) $T.DEPTH \equiv T$
(4) $T.PUSH(a).POP \equiv T$
(5) $\mathcal{E}(T.TOP) \Rightarrow T.TOP \equiv T$

C. Values:

(6) $\mathcal{E}(T) \Rightarrow V(T.PUSH(a).TOP) = a$
(7) $\mathcal{E}(T) \Rightarrow V(T.PUSH(a).DEPTH) = 1 + V(T.DEPTH)$
(8) $V(DEPTH) = 0$

Example 2. An Integer Queue

Syntax:

```
       ADD:   <integer> x <queue> -> <queue>
    REMOVE:               <queue> -> <queue>
     FRONT:               <queue> -> <integer>
```

Legality:

```
    (1)   ℰ(T)  => ℰ(T.ADD(a))
    (2)   ℰ(T)  => ℰ(T.ADD(a).REMOVE)
    (3)   ℰ(T.REMOVE) = ℰ(T.FRONT)
```

Equivalences:

```
    (4)   ℰ(T.FRONT)  => T.FRONT ≡ T
    (5)   ℰ(T.REMOVE)  => T.ADD(a).REMOVE ≡ T. REMOVE.ADD(a)
    (6)   ADD(a).REMOVE ≡ ⊔
```

Values:

```
    (7)   V(ADD(a).FRONT) = a
    (8)   ℰ(T.FRONT) => V(T.ADD(a).FRONT) = V(T.FRONT)
```

The above specification assumes that only one queue exists and omits the queue parameter in the calls on the access programs.

Example 3. Sorting Queue = (SQUEUE)

Syntax:

```
INSERT:  <integer> x <squeue> -> <squeue>
REMOVE:              <squeue> -> <squeue>
 FRONT:              <squeue> -> <integer>
```

Legality:

(1) $\mathcal{E}(T) \Rightarrow \mathcal{E}(T.INSERT(a))$

(2) $\mathcal{E}(T) \Rightarrow \mathcal{E}(T.INSERT(a).REMOVE)$

(3) $\mathcal{E}(T.FRONT) = \mathcal{E}(T.REMOVE)$

Equivalences:

(4) $\mathcal{E}(T.FRONT) \Rightarrow T.FRONT \equiv T$

(5) $T.INSERT(a).INSERT(b) \equiv T.INSERT(b).INSERT(a)$

(6) $INSERT(a).REMOVE \equiv \sqcup$

(7) $\mathcal{E}(T.FRONT)\ \underline{cand}\ (\overline{V}(T.FRONT) \le b) \Rightarrow$
$T.INSERT(b).REMOVE \equiv T$

Values:

(8) $V(INSERT(a).FRONT) = a$

(9) $\mathcal{E}(T.FRONT)\ \underline{cand}\ V(T.FRONT) \le b \Rightarrow$
$V(T.INSERT(b).FRONT) = b$

NOTE the value of X *cand* Y is *false* if X is *false*, and the value of X *cand* Y is the value of Y if X is *true*. Y need not have a defined value if X is *false*.

Example 4. Stack that Overflows (Stac)

Syntax

```
    PUSH:  <stac> x <integer> -> <stac>
     POP:  <stac> -> <stac>
     VAL:  <stac> -> <integer>
```

Legality

For all T, $\mathcal{E}$(T)

Equivalences

$0 < N \leq 124 \Rightarrow \mathrm{PUSH}^{N}(a_i).\mathrm{POP} \equiv \mathrm{PUSH}^{N-1}(a_i)$
$\mathrm{PUSH}(a_0).\mathrm{PUSH}_{i}^{124}.(a_i) \equiv \mathrm{PUSH}_{i}^{124}(a_i)$
$T.\mathrm{VAL} \equiv T$
$N \geq 0 \Rightarrow \mathrm{POP}^{N}.\mathrm{PUSH}(a) \equiv \mathrm{PUSH}(a)$

Values

V(T.PUSH(a).VAL) = a mod 255

Example 5. Alternative Formal Specifications (Guttag Type) for STAC

This alternative includes two "hidden functions," which are marked in the syntactic specifications with asterisk.

TYPE:

```
    stac
```

SYNTACTIC SPECIFICATION:

```
    NEWSTAC:                 -> <stac>
    PUSH(s,I):     <stac> X <integer> -> <stac>
    POP(s):        <stac> -> <stac>
    VAL(s):        <stac> -> <integer>
    SPSLFT(s):     <stac> -> <integer>
   *ADD(s,I):      <stac> X <integer> -> <stac>
   *DEQ(s):        <stac> -> <stac>
```

SEMANTIC SPECIFICATION:

```
    SPSLFT(NEWSTAC) = 124
    SPSLFT(ADD(s,I)) = SPSLFT(s) - 1
    POP(NEWSTAC) = NEWSTAC
    POP(ADD(s,I)) = s
    DEQ(NEWSTAC) = NEWSTAC
    DEQ (ADD(s,I)) = if SPSLFT(s) = 124
                        then s
                        else ADD(DEQ(s),I)
    PUSH(s,I) = if SPSLFT(s) > 0
                        then ADD(s,I)
                        else ADD(DEQ(s),I)
    VAL(NEWSTAC) = undefined
    VAL(ADD(s,I)) = I mod 255
```

*denotes a hidden function

VII. DISCUSSION OF THE SIMPLE EXAMPLES

Example 1 is the classic example for abstract specifications. It is a stack with unlimited capacity. The legality section shows that any sequence of PUSH operations is a legal trace. The first statement in the value section shows the value of TOP after any trace that ends with a PUSH. (7) shows that PUSH always increments the value of DEPTH. (8) specifies the initial value of DEPTH to be zero. The equivalence section allows us to reduce any legal trace with PUSH, TOP, and POP to one that is equivalent but contains only PUSH operations. We will be able to determine the value of the V-functions for any legal trace by making such reductions.

In Example 2 (an integer queue) the "legality" section allows traces that consist of any number of ADDS but each occurrence of REMOVE or FRONT must be preceded directly by an ADD. However, the equivalence statements allow other traces because the sequence ADD.REMOVE may either be replaced by REMOVE.ADD or (at the start of a trace) deleted and the resulting trace will be equivalent to the original trace. The value section shows the value of FRONT after (a) an item is added to an empty queue and (b) an item is added to the queue that already has a value of FRONT (same as before). To find the value of FRONT after a trace that has REMOVES in it, one must apply (5) and (6) repeatedly until one has an equivalent trace that does not contain a REMOVE. Each application of (5) can move a REMOVE to the left one place. When REMOVE follows the first ADD directly, both can be deleted using (6).

In Example 3 we have a queue that always shows the largest item at the front. The largest object is also the one removed by REMOVE. The legal traces are the same as those in Example II (except for an obvious change of function names). The most important difference is (5) in which it is asserted that the order of two consecutive inserts is irrelevant. Assertion (7) shows the effect of a REMOVE after an INSERT that had a parameter larger than the value at the front of the SQUEUE. In that case it simply cancels the effect of the INSERT. However, because of (5), we can always rearrange the order of INSERTs so that the last one is the one that inserts the largest value. This allows us to use (7) for any REMOVE at the end of a trace with at least two inserts in it. (6) describes the effect of REMOVE in the case that it is preceded by only one INSERT. The value section shows us the value of FRONT after an INSERT in an empty queue and after inserting a value that is greater than the value of FRONT.

The discussion of the first three examples is intended to show that the formal specifications do correspond to our intuitive notions of the way that these modules perform.

The correspondence with intuition must, of necessity, remain informal. The demonstration of completeness can be performed systematically. An example of a more systematic proof of completeness will be given later.

The fourth example is the problem that John Guttag could not specify without the use of hidden functions[15]. His specification is included as Example 5. We believe that the brevity of our specification shows the advantages of the trace method. This is a situation in which the values of V-functions for some legal traces are deliberately <u>not</u> defined. Any syntactically correct trace is legal. The module will never "trap." However the value of VAL initially (or after a POP on an "empty stack") is not defined. The implementation can deliver any value in these situations without violating the specifications. If a value, I, greater than 255, is inserted only I mod 255 will be stored.

The above examples show a number of advantages over previous methods of DA specifications. There appears to be no need for hidden functions; the specifications are quite compact and the individual statements are simple. The derivations needed to demonstrate completeness are sometimes quite involved but they need not be performed during the implementation or during the verification that an implementation is correct.

The ideas are rather new and we are aware of a number of important unanswered questions. Nonetheless, we believe that this report demonstrates that the method is as good as any of the peviously published ones and can help to discover design errors early in the design process.

VIII. <u>A COMPRESSED HISTORY OF THE DEVELOPMENT OF AN ABSTRACT SPECIFICATION</u>

In this section we present the history of the development of an abstract specification for a "table/list"-(T/L) module. The programs offered by this module support the processing of linearly ordered data structures, regardless of whether they are implemented as tables or lists. This module is currently implemented to help in generating address translation tables as we need them for a virtual memory mechanism within a family of operating systems (BSF)*[17]. It is also expected that this specification can be used for various other table or list handling purposes.

A. <u>An Informal Picture of the T/L Module</u>

Because it is the purpose of this report to introduce a method of describing such modules, we must begin with an intuitive description of our example. One physical implementation of this module would be by means of a set of children's blocks where it is possible to write one "entry" on the upper surface. The blocks are arranged in a single row and covered with an opaque lid with a single window. Through this window one may read the entry on a single block, insert and remove blocks, or change the entry written on the block that shows through the window. The entry on the block that shows through the window is referred to as the current entry. Because the cover is opaque it is not possible to tell how many blocks are currently under it, but the cover is fitted with signals that tell whether or not there is a block to the right of the current entry, whether or not there is a block to the left of the current entry, and whether there are any blocks under the cover at all.

The operations that we want to perform include reading the value of the current entry, moving the lid one place to the right, moving the lid one place to the left, moving the lid and all blocks to the right of the current block to the right so that a new current block may be inserted through the window, and removing the current block (moving the lid and all blocks to the right of the deleted block one place to the left).

It was our goal that all operations that could be easily performed with the physical model described above be allowed by our specification.

In our specification we will have five operations (O-functions): INSERT,DELETE, ALTER, GOLEFT, and GORIGHT. ALTER will just be a shorthand for a sequence of DELETE and INSERT. The first two indicators mentioned above will be named EXLEFT(EXist entries to the LEFT), EXRIGHT, and the third is represented by EMPTY. The current entry will be available through the V-function CURRENT. The precise relationship among the V-functions and the way that their values are changed by the module's operations will be described in the specifications.

B. The First Version (Example 6)

We do not display the original specification but instead present a translation using traces. We were not using traces for specification purposes at the time that the original was written. The use of traces makes many deficiencies in the first version obvious. They were originally discovered after much hard labor. We show an abbreviated history of the development to provide evidence controverting the claim that abstract specifications state "only the obvious."

Example 6. (Incorrect) Version of a Specification for a Table/List Module

Syntax of Functions

```
O-Functions:  INSERT(e):   <entry> x <TL> -> <TL>
              DELETE:      <TL> -> <TL>
              ALTER(e):    <entry> x <TL> -> <TL>
              GOLEFT:      <TL> -> <TL>
              GORIGHT:     <TL> -> <TL>

V-Functions:  CURRENT:     <TL> -> <entry>
              EMPTY:       <TL> -> <boolean>
              EXLEFT:      <TL> -> <boolean>
              EXRIGHT:     <TL> -> <boolean>
```

Legal Traces

```
 (1)  ℰ(T)  =>  ℰ(T.INSERT(e))
 (2)  ℰ(T)  =>  ℰ(T.INSERT(e).CURRENT)
 (3)  ℰ(T.CURRENT)  =>  ℰ(T.EXLEFT)
 (4)  ℰ(T.CURRENT)  =>  ℰ(T.EXRIGHT)
 (5)  ℰ(T.CURRENT)  =>  ℰ(T.ALTER(e))
 (6)  ℰ(T.CURRENT)  =>  ℰ(T.INSERT(e).GOLEFT)
 (7)  ℰ(T.GOLEFT)  =>  ℰ(T.GOLEFT.GORIGHT)
```

Equivalences

```
 (8)  T.EMPTY ≡ T
 (9)  T.INSERT(e).DELETE ≡ T
(10)  T.GOLEFT.GORIGHT ≡ T
(11)  T.ALTER(e) ≡ T.DELETE.INSERT(e)
(12)  ℰ(T.CURRENT)  =>  (T.CURRENT ≡ T)
(13)  ℰ(T.EXLEFT)  =>  (T.EXLEFT ≡ T)
(14)  ℰ(T.EXRIGHT)  =>  (T.EXRIGHT ≡ T)
```

Values

```
(15)  V(EMPTY)  =  true
(16)  ℰ(T)  =>  (V(T.INSERT(e).CURRENT)  =  e)
(17)  ℰ(T)  =>  (V(T.INSERT(e).EMPTY)  =  false)
(18)  ℰ(T) ∧  (V(T.EMPTY)  =  true)  =>
              (V(T.INSERT(e).EXLEFT)  =  false)
(19)  ℰ(T) ∧  (V(T.EMPTY)  =  false) ^  (V(T.EXLEFT)  =  false) =>
              (V(T.INSERT(e).EXLEFT)  =  true)
(20)  ℰ(T)  =>  (V(T.INSERT(e).EXRIGHT)  =   V(T.EXRIGHT)
(21)  ℰ(T.GOLEFT)  =>  (V(T.GOLEFT.EXRIGHT)  =  true)
(22)  ℰ(T.GORIGHT)  =>  (V(T.GORIGHT.EXLEFT)  =  true)
(23)  ℰ(T.ALTER(e))  =>  (V(T.ALTER(e).CURRENT)  =  e)
(24)  ℰ(T.ALTER(e))  =>  (V(T.ALTER(e).EMPTY)  =  V(T.EMPTY))
(25)  ℰ(T.ALTER(e))  =>  (V(T.ALTER(e).EXLEFT)  =  V(T.EXLEFT))
(26)  ℰ(T.ALTER(e))  =>  (V(T.ALTER(e).EXRIGHT)  =  V(T.EXRIGHT))
(27)  V(T.INSERT(e).GOLEFT.CURRENT)  =  V(T.CURRENT)
(28)  V(T.INSERT(e).GOLEFT.EXLEFT)  =  V(T.EXLEFT)
```

The "syntax" section is as in the earlier examples. We use elements of a type "entry" only to store them into the data structure of the T/L module, or to fetch them. We assume that the relation of equality over entries is defined elsewhere.

Statements (3) through (5) tell us that V-functions EXLEFT and EXRIGHT and O-function ALTER(e) have the same applicability condition as CURRENT.

The "equivalences" section should allow the reader to transform any legal trace to one shown to be legal by (1) through (7). The alert reader will notice that this section does not satisfy this requirement. This will be investigated in some detail later.

Statement (8) is unconditional because a call on EMPTY can always be added to or removed from any trace without making the module trap.

Statements (9) and (10) say that subtraces INSERT(e).DELETE and GOLEFT.GORIGHT have no effect. Statement (11) is supposed to tell us that a call on ALTER has the same effect as two consecutive calls on DELETE and INSERT, provided that INSERT has the same actual parameter as ALTER. Statements (12) through (14) tell us that V-functions CURRENT, EXLEFT, and EXRIGHT can be removed from a legal trace to get an equivalent trace.

Statement (15) gives the initialization of the module. Statements (16) through (20) describe the effects of INSERT at the end of a legal trace on the values of EMPTY, CURRENT, EXLEFT, and EXRIGHT.

Statements (23) through (26) define the effects of ALTER at the end of a trace on the four V-functions. Note that only CURRENT is changed.

Statements (27) and (28) say that two consecutive calls on INSERT and GOLEFT have no effect on the values of CURRENT and EXLEFT.

C. Discussion of Flaws in the First Version of the T/L Module Specification

The use of traces and the way in which the present specifications are divided into sections allows us to discuss flaws in version 1 of the T/L module in a straightforward way and to omit two or three intermediate stages of the original development. However, all errors below were actually included in the original design of the T/L module and allowed to remain in the design after formal discussions among the members of our group.

D. Incompleteness

In examining the first specification we first attempt to make certain that the specification is complete. We will (by definition) consider the specification to be incomplete if there are some traces ending in calls on V-functions which can be shown to be legal but for which no value can be derived.

One example of incompleteness concerns the value of the function EXRIGHT. Only (20) and (26) make any statement about the value of EXRIGHT and these make no statement about the initial value of EXRIGHT or V(INSERT(e).EXRIGHT) which can be shown to be legal.

The specification is similarly incomplete with respect to EXLEFT.

Another form of incompleteness can be found by attempting to derive the value of V(INSERT(a).INSERT(b).GOLEFT.EMPTY). There is no statement about the value of EMPTY when immediately preceded by GOLEFT and no equivalence assertion that would allow us to remove GOLEFT.

E. Specification Versus Intuitive Understanding

In addition to the instances of incompleteness that have been demonstrated, we can show that a number of statements in the "legal trace" section and "equivalences" section do not meet our intuitive expectations. There is a problem with the legality of traces beginning with a call on GOLEFT. For example, we would expect that a call on GOLEFT before the first entry has been inserted into the data structure should not be permitted. However, the value of ℒ(GOLEFT.GORIGHT) can by statement (10) always be calculated to be ℒ(␣), which is (by definition) "true". Since by definition ℒ(t.X) => ℒ(t) we can conclude that (for t = GOLEFT and X = GORIGHT) we have ℒ(GOLEFT) = true. A similar problem exists concerning the legality of traces ending with a call on GOLEFT.

Statements (2) and (6) eliminate the possibility of insertion to the left of the leftmost entry. We can move the slot in our cover over the leftmost enry but not further. An insert would then make EXLEFT true again (statement (19)) but we would have inserted to the right of the leftmost entry.

The mnemonic "EMPTY" was an obstacle to a straightforward solution. Imagine that one moves left from the left end. By statement (18), EMPTY would become true although there are entries in the data structure.

We will eliminate these problems by renaming "EMPTY" to "OUT" and allowing one move to the left beyond the left end. The value of CURRENT is then undefined, while OUT is true, EXLEFT is false, and EXRIGHT is true. This is in contrast to the new initial state (no entries in the data structure) where EXRIGHT is false.

A problem that initiated the development of the specification technique presented in this paper is best formulated by posing the following question.

How can the designer be sure that he specified the effects of all traces that he wants to be executable programs?

Or, put in other way and applied to our example, how do we determine the subset of

{INSERT(e),DELETE,ALTER(e),GOLEFT,GORIGHT,
CURRENT,OUT,EXLEFT,EXRIGHT}*,

(where "*" is the Kleene star) that comprises the set of executable, i.e. legal traces?+

We now note some quantitative properties of such traces: Let |X| denote the number of calls on X in a given trace. Then for all legal traces:

+ Rules for including V-functions are easy to find and are therefore not considered now.

$$|\text{GOLEFT}| \geq |\text{GORIGHT}|$$
$$|\text{INSERT}| \geq |\text{GOLEFT}| - |\text{GCRIGHT}|$$
$$|\text{INSERT}| \geq |\text{DELETE}| + |\text{GOLEFT}| - |\text{GORIGHT}|$$

These relations alone, however, help little. The obviously unreasonable trace

GORIGHT.GOLEFT.GOLEFT.INSERT(a).INSERT(b)

satisfies the above inequalities.

We therefore have to make some additional assertions to characterize the set of legal traces.

The specification of Example 6 did not capture the language of the module, as we intuitively understand it. For example:

$\mathcal{L}$(INSERT(a).INSERT(b).GOLEFT.GCLEFT) = <u>false</u>

Other examples can easily be found.

F. The Current Specification for the T/L Module

After discovering the above errors (over a period of several months) we made an observation that allowed us to write the specification given in Example 7.

Any legal trace for the T/L module must be equivalent to a trace in which there is a (possibly empty) sequence of INSERTs followed by any number of repetitions of the sequence INSERT.GOLEFT. This observation is based on our intuitive model of the object that we are trying to specify. (We have no other possible basis). We could create the table contents a_0, a_1... a_i... a_N, where a_i is the current entry by successivly inserting a_0, a_1... a_i and then executing INSERT (a_j);GOLEFT for j = n, n-1......i+1. Each INSERT(a_j).GOLEFT sequence leaves CURRENT unchanged but inserts a block to the right of current.

We will refer to the traces in this form as <u>normal form</u> traces.

The assertions labeled "legal traces" in Example 7 ((1) - (3)) state that all traces in normal form (and some additional traces) are legal. We also indicate that CURRENT may be called whenever a GOLEFT would be allowed.

The assertions (4) - (7) state that the V-functions do not effect any changes on the module. (8) and (9) give the obvious facts that GOLEFT can be cancelled by a GORIGHT that follows it and that an INSERT can be cancelled by a DELETE

Example 7. Table/List Module with Unlimited Capacity

Syntax

```
O-Functions:  INSERT:   <entry> x <TL> -> <TL>
              ALTER:    <entry> x <TL> -> <TL>
              DELETE:   <TL> -> <TL>
              GOLEFT:   <TL> -> <TL>
              GORIGHT:  <TL> -> <TL>

V-Functions:  CURRENT:  <TL> -> <entry>
              OUT:      <TL> -> <boolean>
              EXLEFT:   <TL> -> <boolean>
              EXRIGHT:  <TL> -> <boolean>
```

Legal Traces

```
 (1)  ℰ(T) => ℰ(T.INSERT(a))
 (2)  ℰ(T) => ℰ(T.INSERT(a).GOLEFT)
 (3)  ℰ(T.GOLEFT) => ℰ(T.CURRENT)
```

Equivalences

```
 (4)  T.OUT ≡ T
 (5)  T.EXLEFT ≡ T
 (6)  T.EXRIGHT ≡ T
 (7)  ℰ(T.CURRENT) => (T.CURRENT ≡ T)
 (8)  ℰ(T.GOLEFT) => (T.GOLEFT.GORIGHT ≡ T)
 (9)  T.INSERT(a).DELETE ≡ T
(10)  T.INSERT(a).GOLEFT.DELETE ≡ T.DELETE.INSERT(a).GOLEFT
(11)  ℰ(T) => (T.INSERT(a).INSERT(b).GOLEFT ≡
         T.INSERT(b).GOLEFT.INSERT(a))
(12)  T.ALTER(a) ≡ T.DELETE.INSERT(a)
```

Values

```
(13)  V(OUT) = true
(14)  V(EXLEFT) = V(EXRIGHT) = false

(15)  ℰ(T) => (V(T.INSERT(a).CURRENT) = a)
(16)  ℰ(T) => (V(T.INSERT(a).OUT) = false)
(17)  ℰ(T) => (V(T.INSERT(a).EXLEFT) = not V(T.OUT))
(18)  ℰ(T) => (V(T.INSERT(a).EXRIGHT) = V(T.EXRIGHT))

(19)  ℰ(T.CURRENT) => (V(T.INSERT(a).GOLEFT.CURRENT) =
         V(T.CURRENT))
(20)  ℰ(T) => (V(T.INSERT(a).GOLEFT.OUT) = V(T.OUT))
(21)  ℰ(T) => (V(T.INSERT(a).GOLEFT.EXLEFT) = V(T.EXLEFT))
(22)  ℰ(T.GOLEFT) => (V(T.GOLEFT.EXRIGHT) = true).
```

that follows it. Note that (8) only applies when GOLEFT is legal.

If our specification is a good one, we should be able to show that every legal trace is equivalent to a trace in normal form. The V-functions can be trivially deleted. We are able to delete a DELFTE if it immediately follows an INSERT and a GORIGHT if it follows immediately after a GOLEFT. Using statement (11) we can move a GOLEFT right or left through a sequence of INSERTs to get an equivalent trace. That will allow us to remove instances of DELETE by bringing an INSERT up to them if only GOLEFTs intervene. Using assertion (10) one may transform sequences containing GOLEFT.DELETE and DELETE.GOLEFT into equivalent sequences where either the DELETE has been moved to the left (bringing it closer to the INSERT that it cancels) or the GOLEFT has been moved to the right (bringing it close to any GORIGHT that would cancel it). Assertion (12) allows the removal of all occurrences of ALTFR. Repeated application of these rules allows the removal of all functions except INSERT and GOLFFT.

G. Completeness of the Current Specification

To demonstrate completeness we examine primarily the value section (13)-(22). (13) and (14) specify the initial values of all V-functions except CURRENT. The failure to specify an initial value for CURRENT is not an instance of incompleteness because CURRENT is not a legal trace. Using (15)-(18) we have specified the values of all four V-functions for traces containing only INSERT.

Using (19)-(22) we can determine the values of the V-functions for any trace of the form T.INSERT( ).GOLEFT provided that we know the values of those functions after T. It follows that we know the values for any trace in the normal form. Since the equivalence statements allow any legal trace to be reduced to an equivalent trace in that form, the specification is complete.

H. Consistency

Demonstraction of consistency is more complex. It is quite clear that the value section ((13)-(22)) is in itself consistent, but it is necessary to show that the transformations allowed by the equivalence section that produce a trace ending in a given V-function result in traces with the same value. Such a proof is beyond the scope of this paper.

IX. CONCLUSION

It is clear that when we entered into the design of the T/L module interface we did not expect the difficulties that we encountered. Each proposal seemed intuitively obvious and the formal specifications that we wrote appeared to correspond to our intuition. Several people examined the specifications (which were written using weakest preconditions); all thought that they were acceptable. The types of difficulties described in connection with the first version of the T/L module specification came as a complete surprise. We had expected that writing the formal specifications was "only a formality" for so simple a module.

Our first conclusion then is simply that writing the formal specifications is useful even for simple modules. Had we been forced to make the change from the first version to the second version after coding was underway, it would have been expensive in terms of the amount of code (both in the module and in programs that use the module) that would have needed revision.

Once we became aware of the difficulties, we found attempts to convince ourselves of the correctness of new versions to be extremely frustrating. The specifications that were written (using predicate transformers for programs consisting of calls on the functions) did not lend themselves well to examination for completeness and consistency. The mathematical model underlying those specifications is complex and there were difficulties intrinsic in the decision to talk about programs rather than traces. Although we have not yet produced a complete formal proof that this specification is complete and consistent, the intuitive justifications are far more convincing than our more formal arguments about the old specifications. Our second conclusion therefore is that the concept seems to be superior to other forms of data abstract specification known to us.

It is becoming popular among software specialists to speak of "front end" investment. The proposal is that by investing time and intellectual energy in the early design phase one can reduce the overall systems costs because of time saved at the later stages. A weakness of the majority of such proposals is that they provide little in the way of specific suggestions about what to do at those early stages. There is little evidence that the effort invested in the early stages will actually pay off. There is lots of evidence that just writing vague statements of good intentions ("The system will have a user-oriented interface") will not pay off. In this paper we have made a specific proposal for the use of that "front end" energy. We have shown how to write such specifications, and described how one may evaluate them for completeness and consistency.

Further work on verifying properties of these specifications is clearly necessary. As Price has shown [10], there are clear advantages to doing as much verification as possible before implementation begins. Similar views are found in [18], but Price included some (machine assisted) proofs.

Acknowledgement

The authors are grateful to Professor D. Stanat for his advice while the research was being performed and on the writing of this paper. Dave Weiss, Lou Chmura, John Shore, and Janusz Zamorski also made helpful comments. This research was supported by the U.S. Army under contract #DAAG 29-76-G-0240. W. Bartussek was also supported by the German Academic Exchange Service (DAAD) under stipend #4-USA-CDN-AUS-NZ-3-EB.

REFERENCES

[1] Parnas, D.L. "Information Distribution Aspects of Design Methodology." Proc. IFIP Congress, 1971.

[2] Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules." Communications of the ACM (Programming Techniques Department), December 1972.

[3] Parnas, D.L., Shore, J.E., and D. Weiss. "Abstract Types Defined as Classes of Variables." Proc. Conference on Data: Abstraction, Definition, and Structure, pp. 22-24, Salt Lake City, Utah, March 1976.

[4] Floyd, R.W. "Assigning Meanings to Programs." In "Mathematical Aspects of Computer Science" (J.T. Schwartz, ed.). Proc. Symp. of Applied Mathematics, Vol. 19, American Math. Society, Providence, 1967, 19-32.

[5] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming." Comm. ACM 12, 10. October 1969, 576-583.

[6] Dijkstra, E.W. "Guarded Commands, Nondeterminancy, and the Formal Derivation of Programs." CACM 18, 8, August 1975.

[7] Dijkstra, E.W. A Discipline of Programming. Prentice Hall, 1976.

[8] Parnas, D.L. "A Technique for Software Module Specification with Examples." Comm. ACM, May 1972.

[9] Guttag, J. "The Specification and Application to Programming of Abstract Data Types." Ph.D. Thesis, CSRG TR 59, University of Toronto, September 1975.

[10] Guttag, J. "Abstract Data Types and the Development of Data Structures." SIGPLAN/SIGMOD Conference on DATA: Abstraction, Definition and Structure (to be published in CACM).

[10] Price, W.R. "Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems." Technical Report (Ph.D. Thesis), Carnegie-Mellon University, June 1973, AD766292.

[11] Parnas, D.L. and W.R. Price. "The Design of the Virtual Memory Aspects of a Virtual Machine." Proceedings of the ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems, March 1973.

[12] Parnas, D.L. and W.R. Price. "Using Memory Access Control as the Only Protection Mechanism." Proc. of International Workshop on Protection in Operating System, 13-14 August 1974, IRIA.

[13] Roubine, O. and L. Robinson. "Special Reference Manual" (Second Edition), Technical Report CSG-45, Stanford Research Institute, Menlo Park, Calif.

[14] Liskov, B. and V. Berzins. "An Appraisal of Program Specifications." Research Direction in Software Technology (P. Wegner, ed.). To be published by MIT Press.

[15] J. Guttag. Private communication, 1976.

[16] Parnas, D.L. and H. Wuerges. "Response to Undesired Events in Software Systems." Proc. of the 2nd International Conference on Software Engineering, 13-15 October 1976, San Francisco, California.

[17] Parnas, D.L., Handzel, G. and H. Wuerges. "Design and Specification of the Minimal Subset of an Operating System Family." Presented at 2nd International Conference on Software Engineering, 13-15 October 1976; published in special issue of IEEE Transactions on Software Engineering, December 1976.

[18] Neumann, P.G., et.al. A Provably Secure Operating System: The System, Its Applications, and Proofs. Final Report, Stanford Research Institute, 11 February 1977, Menlo Park, California

[19] Parnas, D.L. "The Use of Precise Specifications in the Development of Software." Proc. IFIP Congress 1977, North Holland Publishing Company.

[20] Parnas, D.L. and G. Handzel. "More on Specification Techniques for Software Modules." Technical Report, Technische Hochschule Darmstadt, Darmstadt, West Germany, February 1975.