80-014

A PRELIMINARY STUDY FOR THE FORMAL FOUNDATION OF TRACE SPECIFICATIONS

by

John McLean

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

presently with
Computer Science and Systems (Code 7593)
Naval Research Laboratory
Washington, DC 20375

INTRODUCTION

W. Bartussek and D. L. Parnas introduced the "trace method" for specifying software in [1], at least partly, in response to Parnas' earlier observation that there was no "precisely defined notation for writing abstract specifications...that I feel to be useful".[9, p863] However, no formal foundation was given. A formal foundation for the trace method is necessary for (1) any rigorous description of the method, (2) the design of software support for the specification user, (3) the proof of assertions about trace specifications, and (4) the rigorous comparison of the trace method with other methods of abstract specification.

This report contains the most important elements of a formal foundation for trace specifications: a syntax, a semantics, and a set of inference rules for trace specifications. Also included is a proof of a soundness theorem for the rules of inference vis-a-vis the semantics, and sample applications of this theorem to assertions concerning the consistency and completeness of trace specifications. Finally, the method is compared with the algebraic approaches, and the direction for future research is indicated.

This report assumes an understanding of the infomal notion of "traces" as given in [1] and an elementary knowledge of set theory as, e. g., given in [8]. Although no knowledge of formal logic is assumed, some background in logic as, e. g., can be obtained from [7] would be useful.

SYNTAX FOR TRACE SPECIFICATIONS

The first step in formalizing the notion of a trace specification for a module M is to define precisely what such a specification is. Such a definition consists of giving a language L the specification is to be written in and then stating how the well-formed expressions of L can be combined so as to yield a specification.

I. Language for Specifications

L is defined by giving its vocabulary and the formation rules used to combine vocabulary elements into well-formed expressions.

A. Vocabulary

The vocabulary of L consists of parentheses (,); the logical connectives -, &, v, $\rightarrow$, $\leftrightarrow$; the existential quantifier E; the equality symbol =; and the following additional elements.

1. Trace Expression Variables:

   A, B, C, ... possibly superscripted.

2. Trace Expression Constants:

   e is the only trace expression constant.

3. Trace Predicates:

L and $\equiv$

4. Trace Functions:

The dot (.) and V.

5. Access Procedures:

Any procedure supplied by the module to its users is an access procedure. If the procedure returns a value it is a functional procedure (called V-Functions in [1]).

6. Domain Names:

Any name used by the module to refer to a countable, nonempty set of values that a parameter or return value of an access procedure can assume is a domain name. Such domains are said to be named.

7. Domain Constants:

Any member of a named domain is a domain constant. The elements of a domain d are said to be of type d .

8. Domain Variables:

a_d, b_d, c_d, ... possibly superscripted, where d is a domain name. A domain variable of the form α_d is said to be of type d.

9. Procedure Calls:

If proc is an access procedure that takes n parameters, then proc($\underline{a}^1$,...,$\underline{a}^n$) is a procedure call if each $\underline{a}^i$ is a domain constant or domain variable of the appropriate type to serve as proc's ith parameter. Procedure calls to functional procedures are function calls.

B. Formation Rules

1. Syntax Sentences:

A syntax sentence is a string either of the form

proc: (d_1) x...x (d_n) or of the form

proc: (d_1) x...x (d_n) --> (d) where proc is an access procedure. A d_i must be included for each parameter passed to the procedure and names the ith parameter's domain. The second form is used if and only if proc is a functional procedure, in which case d names the domain of the value returned by the procedure.

2. Domain Sentences:

A domain sentence is a string of the form d = D where d is a domain name and D is a description of the domain that specifies the objects in the domain and admissible operations upon them. An axiomatic description of the resulting structure must be available.

3. Variables:

variable -->

domain variable |

trace expression variable

4. Trace Expressions:

trace expression -->

trace expression constant |

trace expression variable |

procedure call |

trace expression.trace expression

(Although not a language primitive, it is suggested that the following notational abbreviation be employed to denote repeated strings of procedure calls. Let s be any nonempty string of procedure calls and let si/n be the string that results from replacing every domain variable of the form $\underline{a}^i$ in s by $\underline{a}^n$. For all m≥n, $\prod_i^m[s]$ is defined as follows: $\prod_i^n[s]$ = si/n and for all m>n $\prod_i^m[s] = \prod_i^{m-1}[s].\prod_m^m[s]$.)

5. Terms:

term -->

domain variable |

domain constant |

V(trace expression)

6. Assertions:

assertion -->

L(trace expression) |

trace expression=trace expression |

term=term |

trace expression=trace expression |

-assertion |

(assertion & assertion) |

(assertion v assertion) |

(assertion --> assertion) |

(assertion <--> assertion) |

(Evariable)assertion

(variable)assertion

Assertions are said to be about the module M. Their parentheses will be dropped from the outside of Boolean expressions for the sake of readability when no ambiguity results. Each occurrence of a variable v in an assertion (v)A or (Ev)A is said to be bound. Occurrences that are not bound are free. An assertion is closed if it contains no free occurrences of any variable.

II. Specification

A trace specification of a module is an ordered pair (syntax specification, semantic specification). A syntax specification is an ordered pair, whose first member is a finite set of syntax sentences corresponding one-to-one with the access procedures for the module, and whose second member is a finite set of domain sentences corresponding one-to-one with every domain over which an access procedure's parameter or return values range. A semantic specification is a recursively enumerable set of closed assertions about the module.

Since all assertions in a trace specification are closed, we can abbreviate assertions of the form (v)A in a specification by A.

SEMANTICS FOR TRACE SPECIFICATIONS

We have yet to assign meanings to the symbols introduced in our trace specification language. This is done by stating under what conditions assertions in the language are true. As such, we must define a model and define what it means for an assertion to be true in a model. This will also allow us to make rigorous at least one sense of the concept of consistency for a trace specification, i. e. having a model, and at least one sense of the concept of sufficient-completeness for a trace specification, i. e. not having models that yield 2 different values for a legal finite string of procedure calls ending in a function call. Although I will further discuss these semantic conceptions of consistency and sufficient-completeness later, it should be clear that there is a model that makes V(T)=a true iff there is an implementation that returns a when accessed by the series of procedure calls T. Hence, semantically inconsistent specifications have no implementation, and specifications that are not sufficiently-complete in this semantic sense have implementations that differ with respect to observable behavior.

Definition of Trace Model

A trace sequence is an ordered pair (D,I) where D is a set of domains and I is a function from syntactic constructs in L to their denotations in D. D

consists of one domain for each parameter and return value domain, and one domain, D_T, which can intuitively be regarded as containing traces. D_T contains the subsets D_L, intuitively containing the legal traces, and D_V, intuititively containing those traces that end in a function call. D_L contains the subset D_e, intuitively containing the null trace, such that $D_e \cap D_V = \emptyset$. Further, D_T is such that $x \notin D_T$ implies that $x \in D_e \cup \{y: y = I[.](u,w)$ for some $u \in D_T$ and $w \in Rng(Rng(I/\{v: v$ is an access procedure$\}))\}$. I meets the following conditions:

(1) $I[=] = \{(x,x): x \in UD\}$

(2) $I[L] = D_L$

(3) $I[V] = f: D_V \cap D_L \rightarrow U(D \sim D_T)$

(4) $I[.] = f: D_T \times D_T \rightarrow D_T$ such that for all x, y, z in D_T

(a) $f(f(x,y),z) = f(x,f(y,z))$

(b) $f(x,y) = x$ if $y \in D_e$

(c) $f(x,y) = y$ if $x \in D_e$

(d) $f(x,y) \in D_T \sim D_V$ if $y \notin D_V \cup D_e$

(e) $f(x,y) \in D_V$ if $y \in D_V$

(f) $f(x,y) \in D_T \sim D_L$ if $x \notin D_L$

(5) $I[\equiv] = \{(x,y)$: x and y meet conditions (a) - (c) below$\}$

(a) $(x,y) \in D_T \times D_T$

(b) for all $z \in D_T$, $I[.](x,z) \in D_L$ iff $I[.](y,z) \in D_L$

(c) for all $z \in D_T \sim D_e$, $I[.](x,z) \in D_V \cap D_L$ iff $I[.](y,z) \in D_V \cap D_L$ and $I[.](x,z) \in D_V \cap D_L \Rightarrow$ $I[V](I[.](x,z)) = I[V](I[.](y,z))$

(6) I[domain name] $\in D \sim D_T$

(7) $I[\alpha_d] \in I[d]$ where α_d is a domain variable or constant.

(8) I[trace expression] is defined as follows:

(a) $I[e] \in D_e$

(b) $I[T] \in D_T$ for any trace variable T

(c) I[procedure call with n parameters] = $I[\underline{proc}](I[\underline{a}_1],\ldots I[\underline{a}_n])$ where proc is the procedure of the call, $\underline{a}_i$ is the ith parameter of the call, and I[procedure which takes n parameters] = f: $(D_{P1},\ldots,D_{Pn}) \rightarrow D^*$ such that D_{Pi} is that element of D associated with the procedure's ith parameter and $D^* = D_V$ if the procedure is a functional procedure, else $D^* = D_T \sim D_V$.

(d) I[T.R] = I[.](I[T],I[R]) where T and R are any trace expressions.

For fixed D, a trace model is the set of all sequences S = (D,I) that are identical except, perhaps, for what I assigns to domain variables, trace variables, and trace expressions containing either of the former.

Definition of truth in a model

Given our definition of trace model, we must now define what it is for such a model to be a model of a particular trace specification. We do this by defining what it is for an assertion to be true in a model.

Consider any trace model M composed of sequences $S_i = (D,I_i)$. Let T and T' be any trace expressions and let t and t' be any terms. Following

Tarski [10], we will define truth in terms of satisfaction.

(1) S_i satisfies L(T) iff $I_i[T] \in I_i[L]$.

(2) S_i satisfies t=t' iff $I^*_i[t]$ and $I^*_i[t']$ are both defined and $I^*_i[t]=I^*_i[t']$ where $I^*_i[x] = I_i[x]$ if x is a term not of the form V(T) and $I^*_i[x] = I_i[V](I[x])$ otherwise.

(3) S_i satisfies $T \subseteq S$ iff $(I_i[T],I_i[S]) \in I_i[\subseteq]$.

For any assertions A and B and any variable v, we employ the standard definition of satisfaction. That is

(1) S_i satisfies -A iff S_i fails to satisfy A.

(2) S_i satisfies A & B iff S_i satisfies A and S_i satisfies B.

(3) S_i satisfies A v B iff S_i satisfies A or S_i satisfies B.

(4) S_i satisfies A $\rightarrow$ B iff S_i satisfies B or S_i fails to satisfy A.

(5) S_i satisfies A $\leftrightarrow$ B iff S_i satisfies both A and B or S_i satisfies neither A nor B.

(6) S_i satisfies (Ev)A iff there is a S_j in M such that S_j satisfies A and I_j is like I_i except perhaps in what it assigns to v and to trace expressions containing v.

(7) S_i satisfies (v)A iff A is satisfied by every S_j in M such that I_j is like I_i except perhaps in what it assigns to v and to trace expressions containing v.

An assertion is <u>true</u> in M iff it is satisfied by every S_i in M.

M is a model for a trace specification iff every assertion in the specification is true in M.

An assertion A is a semantic consequence of a specification S, written $S \models A$, iff A is true in every model of S

TRACE DEDUCTIVE SYSTEM

An alternative to the semantic concept of consistency for a trace specification discussed in the previous section is that one cannot derive $V(\underline{T})=\underline{d}$, $V(\underline{T})=\underline{d}'$, and $-\underline{d}=\underline{d}'$ from the specification for any trace expression $\underline{T}$. An alternative to the semantic concept of sufficient-completeness for a specification is that whenever $L(\underline{T}.\underline{C})$ is derivable from the specification for any finite string of procedure calls $\underline{T}$ and function call $\underline{C}$, then one can derive $V(\underline{T}.\underline{C})=\underline{d}$ for some domain constant $\underline{d}$. In the next section I will examine the relation between these syntactic conceptions of consistency and sufficient-completeness *vis-a-vis* their semantic counterparts. However, we must first formalize these concepts of consistency and sufficient-completeness. This requires a precise definition of derivation.

The following definition of derivation is based on a trace deductive system that has been designed to make derivations relatively easy to construct. Systems that lend themselves more easily to computerized verification of derivations have been constructed from this one by replacing the tautology rule by *modus ponens* and supplementing the axiom set with a complete set of axioms for sentential calculus as can be found, e. g., in [7]. Possible extensions and modifications to the system presented here are discussed at the end of this section. The definition of derivation will refer to axioms and rules of inference as defined below.

Let v be any variable; t any term or trace expression; d any domain variable; P any procedure call; and A, B, and C any assertions. At/s is the result of replacing every free occurrence of t in A by s except where such a substitution would result in a bound occurrence for s.[1] t is of the same type as v if t is a trace variable or expression and v is a trace variable, or if t is a domain variable or constant of type d and v is a domain variable of type d.

Axioms

(1) (Ev)v=t, where v and t are the same type, t not of the form V(T) for any trace expression T.

(2) (v)(A --> B) --> (A --> (v)B), where v is not free in A.

(3) ((v)A & (Ev)v=t) --> Av/t, where Av/t is well-formed.

(4) (Ev)v=t --> t=t

(5) t=t' --> (A <--> A'), where A' is a well-formed assertion and is like A except for possibly having some occurrences of t' where A has t.

(6) T=T.e

(7) T=e.T

(8) T!R <-->

(S)((L(T.S) <--> L(R.S)) &

(-S=e -->

(((Ed)V(T.S)=d <--> (Ed)V(R.S)=d) &

((Ed)V(T.S)=d --> V(T.S)=V(R.S)))))

(9) L(e)

(10) L(T.S) --> L(T)

(11) (Ed)V(T.C)=d <--> L(T.C), where C is a functional procedure call that returns values of the same type as d.

(12) -(Ed)V(T.C)=d, where C is any non-functional procedure call.

(13) -(Ed)V(e)=d

(14) -T=e -->

$$(ES)((Ea_{11})\ldots(Ea_{1n})T{=}S.C_1 \vee \ldots \vee (Ea_{k1})\ldots(Ea_{km})T{=}S.C_k)$$

where C_i is a call on the ith procedure of S with a_{ij} as the jth parameter of the call.

Rules of Inference

(1) Tautology: if A is a tautological consequence (as, e. g., determined by a truth table) of a (possibly empty) set of earlier lines in a derivation, then A may be entered as a line in the derivation.

(2) Universal Generalization: if A appears as an earlier line in a derivation, then one may enter (v)A as a line in the derivation.

(3) Existential Interchange: if A appears as an earlier line in a derivation and B is like A except for having one or more occurrences of (Ev) where A has -(v)- or vice versa, then one may enter B as a line in the derivation.

A derivation from a trace specification S is a finite sequence of lines, each of which is either (1) an assertion contained in the semantic specification of S, or (2) an axiom or an assertion justified by a rule of inference.

An assertion A is derivable from S, written $S \vdash A$, iff there is a derivation from S that has A as a last line

Possible Extensions and Modifications

The above system is minimal in that although I think it correctly represents the traces method as described in [1], it could be extended by the addition of further axioms that reflect modifications to the original description of the method. Such an extension includes stronger axioms for trace equality and axioms that allow for a more radical form of nondeterminism than allowed for here -- i.e. one in which the same implementation may return different values for the same string of function calls at different times.

I chose not to make these extensions part of the system described here for two reasons. First, extensions can be justified only in so far as they improve the traces specification method, and it's not clear to me that the either of these extensions do that. Second, such extensions would lead to a more complicated definition of model than the one presented in the last section and hence, make it harder to verify that a specification has a model. The former extension requires more restrictions to be placed on I[.] so as to render, e. g., $I[.](I[w],I[x]) \neq I[.](I[y],I[z])$ if x and z are distinct procedure calls and hence, also necessitates the inclusion of a new subset D_P of D_T consisting of the denotations of procedure calls. The latter extension requires either the inclusion of temporal elements in the semantics or the elimination of V(T)=V(T) as a theorem of the system unless we introduce

the membership relation into the language and regard the denotation of V(T) as a set or a bunch (as defined in [4]) of return values. Nevertheless, the various ways of implementing these extensions merit discussion.

Axiom (8) requires that if T≢R, then for any nonempty trace expression S such that T.S returns a value, R.S returns the same value. This requirement is incompatable with the inclusion of equivalent, nondeterministic trace expressions unless, as suggested above, we regard such expressions as returning, e. g., sets of values. An alternative would be to replace axiom (8) by one that required merely that T.S and R.S are indistinguishable -- i. e. knowing merely that Q is either T.S or R.S and that Q returns a particular value a is insufficient for concluding that Q is T.S or that Q is R.S. However, we can make the distinction between knowing that Q is T.S or R.S, on one hand, and knowing that Q is T.S or knowing that Q is R.S, on the other, only by vastly supplementing the system, e. g., by adding axioms sufficient to define a proof predicate [2] or an intensional operator [5].

A second property of the system given here is that there are few restrictions on trace equality. Modifications to the system appear below that rule out such possibilities as there being two, finite strings of procedure calls that are equal but not identical.

(1) Change the occurrence of '$\rightarrow$' in axiom (14) to '$\leftrightarrow$'.

(2) Add the following two new axioms:

(i) $\neg \underline{C}_1 = \underline{C}_2$ where $\underline{C}_1$ and $\underline{C}_2$ are any two nonidentical procedure calls.

(ii) $T.\underline{C}_1 = S.\underline{C}_2 \leftrightarrow (T=S \ \& \ \underline{C}_1 = \underline{C}_2)$ where $\underline{C}_1$ and $\underline{C}_2$ are any procedure calls.

SOUNDNESS THEOREM

We have seen a semantic conception of consistency and of sufficient-completeness and we have seen a syntactic conception of consistency and of sufficient-completeness. However, we have yet to bridge the gap between them. The following theorem is fundamental in establishing this bridge.

Theorem: An assertion A is derivable from a specification S only if it is a semantic consequence of S, i. e. $S \vdash A \equiv S \models A$.

Proof: Assume that M is a model of S. We will prove the theorem via induction on the length of A's derivation.

Assume that $S \vdash_m A \Rightarrow S \models A$, for all $m < n$ where $S \vdash_m A$ means that A is derivable from S by a derivation of m steps. We must show that $S \vdash_n A \Rightarrow S \models A$. If the nth line is an assertion contained in S, the proof is trivial since, by assumption, M models S. If the nth line is licensed by an axiom or rule of inference, we have the following possible cases:

(1) If A was inferred by axiom (2) or a rule of inference, then its truth follows by familiar argument from the induction hypothesis since we have employed the standard definition of truth for all connectives and every

assertion in the semantic specification of a trace specification is closed (ruling out possible problems with the application of Universal Generalization).

(2) If A was inferred by axiom (1), then it is of the form (Ev)v=t where v and t are of the same type and not of the form V(trace). This is true in all models given the interpretation of such terms and of identity.

(3) If A was inferred by axiom (3) then it is of the form ((v)P & (Ev)v=t) --> Pv/t where Pv/t is well-formed. Its truth follows from the fact that (Ev)v=t is true if and only if t denotes some object in the domain that is of the same type as v. Given that t is such a term, the axiom reduces to a special case of (v)P --> Pv/t, which is true by standard argument.

(4) If A was inferred by axiom (4) then it is of the form (Ev)v=T --> t=t. The truth of this formula follows from the fact that for every denoting term t, t=t is true.

(5) If A was inferred by axiom (5) then it is of the form t=t' --> (P <--> P') where P' is like P except for some possible occurrences of t'. If t=t' is not satisfied in some sequence, then A is satisfied by that sequence. If t=t' is satisfied by the sequence, then P <--> P', and therefore A, is satisfied by that sequence.

(6) If A was inferred by axiom (6) or (7) then it is of the form T=T.e or T=e.T. By definition of I[.], I[T.e] = I[T] = I[e.T] for any T in any sequence.

(7) If A was inferred by axiom (8) the it is of the form T≒R ⟷ (S)((L(T.S) ⟷ L(R.S)) & (-S=e → (((Ed)V(T.S)=d ⟷ (Ed)V(R.S)=d) & ((Ed)V(T.S)=d → V(T.S)=V(R.S))))). If T≒R is satisfied by a sequence then I[T.S] ∈ D_L if and only if I[R.S] ∈ D_L, and therefore (S)L(T.S) ⟷ L(R.S) will be satisfied by the sequence. Further, for any S that is not the null trace, V(T.S) will be defined iff V(R.S) is and V(T.S) = V(R.S). Therefore, the right hand side of S will be satisfied. If T≒T' is not satisfied by S then one of the above condition must fail making the right hand side not satisfied as well. Hence A is true.

(8) If A was inferred by axiom (9) then it is of the form L(e). But for any model this is true since I[e] is contained in I[L].

(9) If A was inferred by axiom (10) then it is of the form L(T.S) → L(T). This is true on all models by the definition of I[.].

(10) If A was inferred by one of axioms (11)-(13) then its truth follows from the fact that V is defined on all and only legal traces that are in D_V and the fact that any trace expression ending in a function call is in D_V while neither the empty trace nor any trace expression ending in a non-functional procedure call is in D_V.

(11) If A was inferred by axiom (14) then it is of the form -T=e → (ES)(Ea_{11})...(Ea_{1n})T=S.C_1 v ... v (Ea_{k1})...(Ea_{km})T=S.C_k) where C_i is a call on the ith procedure of S with a_{ij} as the jth parameter of the call. Its truth can be seen by noting that if an element of D_T is not the empty trace expression, then it must be equal to I[.](u,w) for some w ∉ Rng(Rng(I/{v: v is a procedure})) where u may be the empty trace.

Corollary: A trace specification is syntactically consistent if it is semantically consistent.

Proof: If a trace specification is not syntactically consistent, then it is easy to see that there must be some trace expression T such that (Ex)V(T)=x & -V(T)=V(T) is derivable from the specification. By the Soundness Theorem, this assertion must be a semantic consequence of any model of the specification. But since the assertion is false in all models, the specification can have no models.

Corollary: A trace specification is sufficiently-complete in the syntactic sense only if it is sufficiently-complete in the semantic sense.

Proof: If a trace specification is not sufficiently-complete in the semantic sense, then there is some finite string of procedure calls T ending in a function call and domain constants a and b such that $a \neq b$ yet I[V](I[T]) = I[a] in one model and I[V](I[T]) = I[b] in another. By the definition of legality in [1], L(T) must be derivable, yet by the soundness theorem, there can be no domain constant d such that V(T)=d is derivable since I[d] would have to be equal to both I[a] and I[b] while $I[a] \neq I[b]$.

It is an open question as to whether the system is complete, i. e. whether the converse of the theorem (and of the theorem's corollaries) holds. This point is further discussed below in the section on future research.

APPLICATIONS

Application to Consistency Proofs

By the first corollary of the previous section one can prove a specification consistent (in either sense of the term) by giving it a model. As an example it can easily be shown that the following stack module is consistent when supplemented by first order number theory.

Stack Specification:

a and b are assumed to be a type integer, while r and s are assumed to be of type name.

Syntax:

PUSH: (int) x (name)

POP: (name)

TOP: (name) --> (int)

DEPTH: (name) --> (int)

int = the set of integers

name = the set of finite character strings

Semantics:

(1) L(T) --> L(T.PUSH(a,s))

(2) L(T.TOP(s)) <--> L(T.POP(s))

(3) T.DEPTH(s)≠T

(4) T.PUSH(a,s).POP(s)=T

(5) -r=s --> T.PUSH(a,s).PUSH(b,r)≠T.PUSH(b,r).PUSH(a,s)

(6) L(T.TOP(s)) --> T.TOP(s)=T

(7) L(T) --> V(T.PUSH(a,s).TOP(s))=a

(8) L(T) --> V(T.PUSH(a,s).DEPTH(s))=V(T.DEPTH(s))+1

(9) (L(T) & -r=s) --> V(T.PUSH(a,s).DEPTH(r))=V(T.DEPTH(r))

(10) V(DEPTH(s))=0

Stack Model:

Note that only those aspects of the model that are invariant across sequences of the model must be given in order to specify it uniquely. Let s be any name variable or constant and i any integer variable or constant. D = int U name U D_T where D_T = {x: x is a possibly empty string of procedure calls without variables}.

I[e] = the empty string

I[t] = t if t is an integer, a name, or a procedure call involving no variables. If the call contains variables, I[t] is the call once each variable v has been replaced by I[v].

I[L] = {x: x is in D_T and such that to the left of every POP(s) and TOP(s) in x there are more PUSH(i.s)'s than POP(s)'s}.

I[V] = a function f from those elements of I[L] that end in TOP or DEPTH, to the integers such that for every $x \in Dom(f)$, $f(x) = n$ if (1) x ends in DEPTH(s), and n is the number of PUSH(i,s)'s in x minus the number of POP(s)'s in x, or (2) x ends in TOP(s) and, scanning x from right to left, PUSH(n,s) is the first occurrence of a PUSH(i,s) in x that cannot be paired with a previous, unpaired POP(s).

I[.] is the concatenation function

$I[\stackrel{\cdot}{=}] = \{(x,y)$: x,y $\in D_T$ and x and y are identical except perhaps in the order of their procedure calls after both x and y have been subjected to the following procedure$\}$:

1. Remove all DEPTH's.
2. Remove every TOP that is such that the initial string of the trace up through it is an element of I[L].
3. Remove the first and last call of all strings of the form PUSH(i,s)...POP(s), where ... is any (possibly empty) string of procedure calls that contains neither POP(s) nor PUSH(i,s) for any i.
4. Repeat #3 as long as possible.

Application to Completeness Proofs

The second corollary to the soundness theorem can be employed to prove that a specification S is not sufficiently-complete (in both senses) if we give two models M_1 and M_2 for S and show that there is a value trace A and domain constant $\underline{a}$ such that $V(A) = \underline{a}$ is true in M_1 and false in M_2. As an example, I will prove the following keysort specifications, adapted from

a similar one suggested by David Parnas, incomplete[2]:

Keysort Specification:

It is assumed that a, a', b, b', x, and y are of type integer.

Syntax:

INSERT: (int) x (int)

REMOVE:

FRONT: --> (pair)

int = the set of integers

pair = $\{(x,y)$: x $\in$ int & y $\in$ int$\}$

Semantics[3]:

(1) L(T) --> L(T.INSERT(a,b).REMOVE)

(2) L(T.FRONT) <--> L(T.REMOVE)

(3) L(T.FRONT) --> T.FRONT$\exists$T

(4) V(T.INSERT(a,b).FRONT)=(a,b) -->

(T.INSERT(a,b).REMOVE$\exists$T v

(V(T.FRONT)=(a,b) &

T.INSERT(a,b).REMOVE?T.REMOVE.INSERT(a,b)))

(5) -V(T.INSERT(a,b).FRONT)=(a,b) -->

T.INSERT(a,b).REMOVE$\exists$T.REMOVE.INSERT(a,b)

(6) V(INSERT(a,b).FRONT)=(a,b)

(7) V(T.FRONT)=(a,b) -->

(V(T.INSERT(a',b').FRONT)=(x,y) -->

((a<a' & x=a & y=b) v

(a>a' & x=a' & y=b') v

(a=a' & x=a & (y=b v y=b'))))

Interpretation:

M_1 and M_2 agree on the following:

(1) D, D_T, I[e], I[.] and I[t] where t is an integer ordered pair or procedure call are analogous to the model for the stack specification.

(2) $I[L] = \{x: x \in D_T$ and such that to the left of every REMOVE or FRONT in x there are more INSERT's than REMOVE's$\}$.

The interpretation of and V depend on the following normalizing algorithm which takes as input strings of procedure calls:

NORMAL(<u>string</u>):

1. If <u>string</u> $\notin D_L$, then abort.
2. Remove each occurrence of FRONT from <u>string</u>.
3. Label the ith INSERT from the left and the jth REMOVE from the left in <u>string</u> $INSERT_i$ and $REMOVE_j$ respectively. Call the key and the ordered pair associated with $INSERT_i$, key_i and $pair_i$ respectively.
4. For k=1 to the number of REMOVE's in <u>string</u>, eliminate the pair $INSERT_j$ $REMOVE_k$ such that the following conditions are met:
   (a) $INSERT_j$ is to the left of $REMOVE_k$ in <u>string</u>.
   (b) If $INSERT_i$ is to the left of $REMOVE_k$, then $key_i \geq key_j$.
   (c) $i>j \Rightarrow$ (a) or (b) fails for $INSERT_i$.

(3) I[V] = a function f from those strings in I[L] that end in FRONT to ordered pairs of integers (a,b). f(T) = (a,b) iff when after the rightmost FRONT of T has been replaced by REMOVE, (a,b) = $pair_i$ where $INSERT_i$ is the last INSERT to be removed when the modified T is subjected to NORMAL.

(4) $I[\Xi] = \{(x,y)$: $x,y \notin D_T$ and when x and y are subjected to NORMAL, either NORMAL aborts for both of them or NORMAL(x) = NORMAL(y) after NORMAL(x) and NORMAL(y) have been sorted into ascending order such that $INSERT_i < INSERT_k$ if $key_i < key_j$ or if $key_i = key_j$ & $i < j\}$.

For M_2, the interpretation of Ξ and V are as above, but with step (4) of NORMAL changed as follows:

4'. For k=1 to the number of REMOVE's in <u>string</u>, eliminate the pair $INSERT_j$ $REMOVE_k$ such that the following conditions are met:

(a) As before.

(b) As before.

(c) If there is an i less then j such that $key_i = key_j$, then there is an n less than j such that $pair_n = pair_j$ and for all m less than n $key_m > key_j$

(d) $i > j \Rightarrow$ (a), (b), or (c) fail for $INSERT_i$

To see that the keysort specification is incomplete one must merely note that if we consider A = INSERT(1,5).INSERT(1,6).FRONT then V(A) = (1,6) in M_1 and V(A) = (1,5) in M_2.

COMPARISON WITH THE ALGEBRAIC APPROACH

The reader will notice two differences between the trace method for specifying software modules and the more algebraic approaches as epitomized, for example, by Guttag and Horning [3]. The first is that the so-called "type of interest" or TOI is never mentioned in a trace specification. For example, within the stack specification, the word "stack" is never used. As such, the specification corresponds more closely to how the user actually sees a stack module, viz. a set of access procedures with certain properties, than the algebraic method which gives relations between the possible "values" stacks can assume. Treating stacks as values renders it necessary to provide error values for stacks to assume and to provide unnecessary functions in order to start, i. e., map an empty value space to an initial stack. Further, by obliterating the distinction between a function call and the value returned by that call, the algebraic approach renders it impossible to represent a sequence such as $call_1.call_2$ except by treating $call_1$ as a parameter of $call_2$. Hence, procedures that affect the "inner state" of the module without returning a value (called O-Functions in [1]) and procedures that do not take parameters cannot be represented in a natural manner by the algebraic approach.[4] Finally, the absence of a TOI within the specification helps to insure a totally behavioristic specification of the module. The mere inclusion of the word "stack" within the specification is a step toward

suggesting implementation since it suggests that for each stack name there must be some object in the implementation, for example an array, that corresponds to that stack. In certain environments, however, it may be more desirable to have only one data structure which stores names and values together.

The second difference between the two methods concerns the languages involved. The trace method makes free use of first order logic with identity while most algebraists prefer more restrictive languages. As such, the trace method allows for much more expressive power. An example is the use of the existential quantifier in axiom (11) to say that any legal trace expression ending in a function call must return some value without saying what that value is. This allows, e. g., for the specification of an integer generating module whose only restriction is that it returns a different integer each time it is called. Such a module can be specified by the single syntax sentence GEN: --> (int) coupled with the two assertions L(T) and -S=e --> -V(R.S)=V(R). The reader should find it enlightening to try to specify this same module algebraically since he will run into problems, not only in trying to capture the nondeterminism of the module, but also, as discussed above, in trying to represent sequences of function calls. Although the richness of the trace language implies that consistency and sufficient-completeness will be harder to establish with the trace method, nobody has found a sufficiently rich language for which consistency and completeness are decidable. Further, the soundness proof given here is an important step toward coming up with a uniform method for establishing trace specifications consistent. The next

step is described as an area for future research.

In spite of the differences between the two approaches, the generality of the term "algebraic" will allow for the development of algebraic models that are formally equivalent to the trace method.[5] The algebraic approach stimulated the development of the trace approach, and the two approaches are quite similar when compared to such alternatives as the "operational definition" and the "abstract model" approaches [6]. Time and energy should not be lost in a debate between the two approaches while practitioners await usable specification tools.

FUTURE RESEARCH

Future research in the trace method can take various forms. First of all, the desirablity and feasability of extending the model so as to allow, e. g., more nondeterminism in specifications and stricter identity conditions between trace expressions should be explored. Second a completeness theorem that every consistent set of assertions has a model must be proven.[6] This theorem will enable research to begin on a model theoretic approach to proving specifications sufficiently-complete and when coupled with the soundness theorem, will complete the bridge between the semantic concepts of consistency and sufficient-completeness and their syntactic counterparts, thus justifying both of them. Third, alternative methods for proving specifications consistent and sufficiently-complete should be studied, a project I am currently engaged in. Fourth, methods for proving the correctness of implementations and the correctness of programs using modules must be developed. Fifth, software support should be developed in order to check specifications for well-formedness, consistency, and sufficient-completeness, as well as to provide quick implementations of specifications. This will allow for the development of much more complex programs, especially where correct specifications are necessitated either by financial reasons or by security reasons. A pilot project to develop such software was undertaken at UNC. Finally, the notation should be extended to allow for more compact and readable specifications.

Questions of a more theoretical nature stem from the inability to say certain things within first order logic. For example, it is impossible to force axiomatically every trace expression variable to denote only finite strings of procedure calls or when dealing with trace expression variables that do denote infinite strings of procedure calls, to restrict equality so that such expressions are equal only if they are identical. Since higher order logics are incomplete, this lack of expressive power must be endured. Nevertheless, issues concerning practical consequences of this fact should be explored.

ACKNOWLEDGEMENTS

The influence of David Parnas' work on this paper is obvious. He has also provided helpful criticism of an earlier draft of this paper, as have David Weiss, Mila Majster, and Donald Stanat. Karen Dwyer, who led the pilot team for developing software support mentioned earlier, pointed out several mistakes in an earlier draft, and Mark Nixon provided some very useful conversation along the way.

FOOTNOTES

1. The notion of occurrence used here is the standard one as used, e. g., in [7] extended so as to regard trace expressions that occur within other trace expressions as occurring in any assertion that the latter occurs in.

2. It should be noted that the incompleteness is deliberate in order not to specify what the module does if duplicate keys appear, beyond stating that such keys are allowed and precede all pairs with greater keys.

3. Strictly speaking, assertions (4) - (7) of this specification are ill-formed since (a,b) is not a variable. Rigor can be maintained by treating the assertion V(T.FRONT)=(a,b) as an abbreviation for FIRST[V(T.FRONT)]=a & SEC[V(T.FRONT)]=b where FIRST[(x,y)]=x and SEC[(x,y)]=y.

4. On the other side of the coin, it should be noted that the trace method makes a sharper distinction between function calls and the values they return than do many programming languages. As such the trace method cannot represent calls that take as a parameter the return value of another call as naturally as the algebraic method.

5. First order logic is, after all, a cylindric algebra.

6. It should be noted that I have recently proven a system that is slightly stronger than the one presented here complete.

BIBLIOGRAPHY

[1] Bartussek, Wolfram and Parnas, David L. Using Traces to Write Abstract Specifications for Software Modules, UNC Technical Report #TR 77-012 (1977).

[2] Gödel, Kurt. "Uber formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme, I", Monatshefte für Mathematik und Physik, XXXVIII (1931), pp. 179-98.

[3] Guttag, John and Horning, J. "The Algebraic Specification of Abstract Data Types," Acta Informatica, X (1978), pp. 27-52.

[4] Hehner, Eric. Simple Set Theory for Computing Science, CSRG Technical Report #102 (1979)

[5] Hughes, G. E. and Cresswell, M. J. An Introduction to Modal Logic (Norwich 1968).

[6] Liskov, Barbara and Berzins, Valdis. "An Appraisal of Program Specifications", Research Directions in Software Technology, ed. P. Wegner (Cambridge, Massachusetts 1979).

[7] Mates, B. Elementary Logic, 2nd ed. (New York 1972).

[8] Monk, J. Introduction to Set Theory (New York 1969).

[9] Parnas, David L. "The Use of Precise Specifications in the Development of Software", Information Processing 77, ed. B. Gilchrist (New York 1977).

[10] Tarski, A., "The Concept of Truth in Formalized Languages", reprinted in Logic, Semantics, Metamathematics (Oxford 1956).