AD A111843

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Memorandum Report 4874	2. GOVT ACCESSION NO. AD-A118 843	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A FORMAL FOUNDATION FOR THE TRACE METHOD OF SOFTWARE SPECIFICATION		5. TYPE OF REPORT & PERIOD COVERED Interim report on a continuing NRL problem. 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) J. McLean		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Washington, DC 20375		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N; RR014-09-41; 75-0228-0-2
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE September 1, 1982 13. NUMBER OF PAGES 62
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Abstract specification Computer software		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) An overview of the trace method for the abstract specification of software is followed by a syntax, semantics, and derivation system for the method. This foundation supports a comparison of the trace method with the algebraic approaches to abstract specification, while completeness and soundness theorems suggest methods for proving specifications consistent and sufficiently-complete. Areas for future research are discussed.		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CONTENTS

PREFACE

W. Bartussek and D. L. Parnas introduced the "trace method" for abstract software specification in [1], at least partly, in response to Parnas' earlier observation that there is no "precisely defined notation for writing abstract specifications...that I feel to be useful" [13, p863]. The method is useful, but since it has received no formal foundation, it falls short of Parnas' goal of being "precisely defined". 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 their implementations, and (4) the rigorous comparison of the trace method with other formal methods of abstract specification.

This report contains an informal overview of the trace method and abstract specification techniques in general, followed by a formal foundation for the method: a syntax, a semantics, and a set of inference rules for trace specifications. Also included is a proof of a soundness theorem and of a completeness theorem for the rules of inference vis-a-vis the semantics, and sample applications of these theorems to assertions about the consistency and sufficient-completeness of trace specifications. Finally, the method is compared with the algebraic approaches to abstract specification, and areas for future research are discussed.

Strictly speaking, the report is self-contained with respect to both formal logic and the trace method. Nevertheless, some background in logic, as, e. g., can be obtained from [10], would be useful, as would an informal understanding of the trace method, as presented in [1]. An elementary knowledge of set theory, as, e. g., given in [11], is assumed.

A FORMAL FOUNDATION FOR THE TRACE METHOD OF SOFTWARE SPECIFICATION

INTRODUCTORY OVERVIEW OF THE TRACE METHOD

The trace method is a formal method for the abstract specification of software, where "software" is liberally construed to cover any program (procedure) or set of programs. As such, in so far as the terms "abstract data type" and "module" are used to refer to sets of related programs, it is a method for their specification as well. Hence, I will freely interchange these terms.

Being formal does not distinguish the trace method from a large number of other methods for specifying software. However, being abstract does. By "abstract" I mean that a trace specification describes only those features of a program that are essential. In particular it is a totally behavioristic specification: it specifies what the program does without describing a method for doing it. If the use of a particular algorithm is required, then this is included as a constraint to, not as a part of, the specification (see [6] for a discussion of the problem of presenting algorithmic constraints in a requirements document). In this respect, it differs from procedure specification methods that are based on "operational definitions" and abstract data type specification methods that are based on "abstract models" [9]. For our purposes, the most important feature of these latter classes of methods is that they specify software by giving a paradigm implementation.

Manuscript submitted May 26, 1982.

The advantages of abstract specification are thoroughly discussed in [13], but it is worthwhile reviewing them here. First, abstract specifications do not contain clutter. This property makes specifications more perspicuous and easier to handle in proofs about implementations. Most importantly their freedom from clutter eliminates a whole class of potential misunderstandings: those that result from the attempted gleaning of the essential features of a specification from a mass of extraneous details. Second, abstract specifications are conducive to good programming practice. By requiring program output to be specified solely in terms of input, the trace method not only forces designers to make any information shared by two or more modules part of an explicit interface, it also discourages unnecessary modular coupling by focusing the designer's attention on such shared information. This makes independent implementation of modules possible and leads to understandable software that is easier to maintain [12].

With respect to the trace method in particular, programs are specified by describing three properties they possess:

(1) What do the access procedures of the programs look like, i. e., what are their names, their parameter types, and their return values types if any? These properties are indicated by sentences of the form $\underline{\text{proc}}$: $(\underline{\text{parm}_1\text{type}})$ x...x $(\underline{\text{parm}_n\text{type}})$ $\rightarrow$ $(\underline{\text{return value type}})$.

(2) Which series of procedure calls are legal, i. e., are not regarded as being in error? These are indicated by assertions of the form L($\underline{\text{series}}$).

(3) What is the output of legal series of procedure calls that end in a function call? This value is denoted by V($\underline{\text{series}}$).

In order to make specifications more readable, the following abbreviational device is used. If two series of procedure calls agree on legality and return value with respect to future program behavior, then we say they are equivalent and write $\underline{\text{string}}_1 \equiv \underline{\text{string}}_2$.

As an example, consider the following specification of a stack module that contains three procedures: PUSH takes an integer as a parameter but returns no value; POP neither takes a parameter nor returns a value; and TOP takes no parameters but returns an integer. The syntax of the module is specified thus.

```
PUSH: (int)
POP:
TOP: --> (int)
```

The semantics of the module consists of five assertions describing the module's behavior: (1) if a series of preceding procedure calls has not resulted in an error, then PUSH can be legally called with any integer parameter; (2) calling TOP will not result in an error if and only if calling POP does not; (3) calling PUSH followed by POP will not affect the future behavior of the module; (4) if TOP can be legally called, then calling it will not affect the future behavior of the module; and (5) the value of any legal series of procedure calls ending in PUSH followed by TOP is the parameter of that PUSH. These assertions are symbolized thus.

(1) L(T) $\rightarrow$ L(T.PUSH(i))

(2) L(T.TOP) $\leftrightarrow$ L(T.POP)

(3) T≡T.PUSH(i).POP

(4) L(T.TOP) $\rightarrow$ T≡T.TOP

(5) L(T) $\rightarrow$ V(T.PUSH(i).TOP)=i

A formal explanation of exactly how assertions (3), (4), and (5) combine to force stack-like behavior necessitates a formal presentation of the method.

SYNTAX FOR TRACE SPECIFICATIONS

The first step in formalizing the notion of a trace specification is to precisely define the term "trace specification". Such a definition consists of giving a specification language L and 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. It is the smallest set that both contains its vocabulary and is closed under its formation rules. We assume that certain well-specified, countable, nonempty domains are given.

A. Vocabulary

The vocabulary of L consists of parentheses (,); the logical connectives -, &, v, $\rightarrow$, $\leftrightarrow$ (read not, and, or, if then, and if and only if, respectively); the existential quantifier E; the equality symbol =; and the following additional elements.

1. Trace Expression Variables:

A, B, C, ... are each a trace expression variable. They can be superscripted.

2. Trace Expression Constants:

e is the only trace expression constant. (It denotes the empty trace.)

3. Trace Predicates:

The unary predicate L and the binary predicates F and $\equiv$ are each a trace predicate.[1] (F is true of a (D,T) if and only if T returns a value of type D.)

4. Trace Functions:

The dot (.) and V are each a trace function.

5. Procedure Names:

Any finite character string is a procedure name.

6. Domain Names:

The name of any given domain is a domain name. Such domains are said to be named.

7. Domain Constants:

c_d, where c denotes any member of a named domain d, is a domain constant.

8. Domain Functions:

f_d, where f denotes any well-specified function on the members of a named domain d, is a domain function. If f denotes an n-placed function, then f is n-ary. If f denotes a function from d to some domain d', then f is said to be d'-valued.

9. Domain Relations:

R_d, where R denotes any well-specified relation on the members of a named domain d, is a domain relation of type d. If R denotes a relation on n elements, it is _n-ary_.

10. _Domain Variables_:

a_d, b_d, c_d, ..., where d is a domain name, are each a _domain variable_. They can be superscripted.

B. _Formation Rules_[2]

1. _n Place Argument List_:

0 place argument list -->

1 place argument list --> _domain element_

m place argument list -->

m-1 place argument list, _domain element_

for m>1

If each element of an argument list is of the form α_d, the argument list is said to be of _type_ d.

2. _n Place Domain List_:

0 place domain list -->

1 place domain list --> (_domain name_)

m place domain list -->

m-1 place domain list x (_domain name_)

for m>1

3. _Syntax Sentences_:

syntax sentence -->

procedure name: _n place domain list_ |

procedure name: _n place domain list_ --> (_domain name_)

When the syntax sentence is of the latter form, then the procedure name is said to be a <u>function name</u> of <u>type</u> d where d is the rightmost domain name in the sentence. If the procedure name is followed by an n place domain list, it is said to <u>take n parameters</u>. Any domain name that occurs in a syntax sentence is a <u>parameter domain</u>. Any parameter domain that occurs to the right of an arrow is a <u>return value domain</u>.

4. <u>Procedure Calls</u>:

```
    procedure call -->
        procedure name |
        procedure name(n place argument list)
            where n>0
```

5. <u>Variables</u>:

```
    variable -->
        domain variable |
        trace expression variable
```

6. <u>Trace Expressions</u>:

```
    trace expression  -->
        trace expression constant |
        trace expression variable |
        procedure call |
        trace expression.trace expression
```

7. <u>Domain Elements</u>:

```
    domain element -->
        domain constant |
```

domai variable

8. Terms:

term →

domain element |

trace expression |

V(trace expression) |

n-ary domain function(n place argument list)

where the function and argument list are of the same type and $n>0$

Terms have the following types. Domain elements of the form α_d are of type d. Trace expressions are of type "trace expression". Terms of the form f(argument list) where f is a d-valued domain function are of type d. Terms of the form V(trace expression) are untyped. Two terms are of compatable type if both are of type α, both are untyped, or one is of type d where d is a domain name and the other is untyped.

9. Assertions:

assertion →

L(trace expression) |

F(domain name,trace expression) |

trace expression≡trace expression |

n-ary domain relation(n place argument list) |

where the relation name and argument list are of the same type and $n>0$

term=_term_ |

where the terms are of compatable type

-_assertion_ |

(_assertion_ & _assertion_) |

(_assertion_ v _assertion_) |

(_assertion_ --> _assertion_) |

(_assertion_ <-> _assertion_) |

(E_variable_)_assertion_ |

(_variable_)_assertion_

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_ is an ordered pair (syntax specification, semantic specification). A syntax specification is a finite set of syntax sentences. A semantic specification is a recursive set of assertions that contain a procedure call of the form _proc_ (or _proc_($p_1,\ldots,p_n$)) only if the syntax specification contains a sentence of the form _proc_: [--> (d)] (or _proc_: (d_1) x...x (d_n) [--> (d)] where for each i, p_i is a domain element of type d_i).

A trace specification is <u>proper</u> if every assertion in its semantic specification is closed. Since all assertions in a proper trace specification are closed, we can abbreviate assertions of the form (v)A by A. For the rest of this paper, we will mean by "trace specification" proper trace specification, unless explicitly stated otherwise.

SEMANTICS FOR TRACE SPECIFICATIONS

We have yet to assign meanings to the symbols introduced in our trace specification language. This is necessary for the unambiguous interpretation of a trace specification. It is accomplished 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 support rigorous definitions of 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 differ with respect to the values they yield for a legal, variable-free trace expression ending in a function call. Although I will further discuss these semantic conceptions of consistency and sufficient-completeness and the relation between models and implementations later, it should be clear that there is a model that makes $V(\underline{T})=\underline{a}$ true if there is an implementation that returns $\underline{a}$ when accessed by the series of procedure calls $\underline{T}$. Hence, semantically inconsistent specifications have no implementation, and specifications that are sufficiently-complete in this semantic sense do not 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 disjoint tuple of domains and I is a function from syntactic constructs in L to their denotations in D. $D = (D_T, D_1, \ldots, D_n, D_{n+1}, \ldots, D_m)$ where D_T can intuitively be regarded as consisting of (series of) procedure calls ("traces"), D_1-D_n can intuitively be regarded as return value domains, and D_{n+1}-D_m can intuitively be regarded as parameter domains that are not return value domains. D_T contains the subsets D_L, intuitively contisting of the legal traces, and D_V, intuititively contisting of those traces that end in a function call. D_V is partitioned into subsets D_{Vi} where D_{Vi} can intuitively be regarded as consisting of those traces that return values of type d where $I[d]=D_i$. D_L contains the subset D_e, intuitively contisting of the null trace, such that $D_e \cap D_V = \emptyset$. D_T contains only the null trace and traces that are formed from elements of D_T by composing them with a procedure call.[3] More formally, $x \in D_T$ implies that $x \in D_e \cup \{y: y = I[.](u,w)$ for some $u \in D_T$ and $w \in \mathrm{Rng}(\mathrm{Rng}(I/\{v:$ v is a procedure name$\}))\}$, where I meets the following conditions:

(1) $I[=] = \{(x,x): x \in \cup D\}$.

(2) $I[L] = D_L$.

(3) $I[F] = \{\{D_i\} \times D_{Vi}: 1 \leq i \leq n\}$.

(4) $I[V] = f: D_{Vi} \cap D_L \rightarrow D_i,\ 1 \leq i \leq n$.

(5) $I[.] = f: D_T \times D_T \rightarrow D_T$ 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) $f(x,y) \in D_{Vi}$ if $y \in D_{Vi}$,

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

(6) $I[\equiv] = \{(x,y)$: 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 $I[.](x,z) \in D_V \cap D_L \Rightarrow I[V](I[.](x,z)) = I[V](I[.](y,z))$.

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

(8) $I[\alpha_d] \in I[d]$ where α_d is a domain element.

(9) $I[f_d]$ is a function from $I[d]^n$ to $I[d']$ where f is a n-ary, d' valued domain function.

(10) $I[R_d] \subset I[d]^n$ where R is a n-ary domain relation.

(11) 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[proc]$(I[\underline{a}_1],\ldots I[\underline{a}_n])$ where proc is the procedure name of the call, $\underline{a}_i$ is the ith parameter of the call, and I[procedure that 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 $D^* = D_{Vi}$ if the procedure is a functional procedure of type d and $I[d]=D_i$, 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 <u>trace model</u> 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 variables.

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, R any relation name, d any domain name, and t and t' be any terms. Following Tarski [14], 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 F(d,T) iff $(I_i[d], I_i[T]) \in I_i[F]$.

(3) S_i satisfies $R(n_1,\ldots,n_m)$ iff $(I[n_1],\ldots,I[n_m]) \in I[R]$.

(4) 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 domain element or trace expression, $I^*_i[x] = I_i[V](I[T])$ if x is of the form V(T), and $I^*_i[x] = I_i[f](I[n_1],\ldots,I[n_m])$ if x is of the form $f(n_1,\ldots,n_m)$.

(5) S_i satisfies $T \equiv S$ iff $(I_i[T], I_i[S]) \in I_i[\equiv]$.

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

(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 --> B iff S_i satisfies B or S_i fails to satisfy A.

(5) S_i satisfies A <--> 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 _true_ in M if and only if it is satisfied by every S_i in M.

A specification S is _satisfiable_ if there is a sequence that satisfies every assertion in the semantic specification of S.

M is a _model for a trace specification_ if and only if every assertion in the specification is true in M.

An assertion A is a _semantic consequence_ of a specification S, written

$S \not\vdash A$, if and only if A is true in every model of S.

Although no reference to programs is made in the definition of "model", it should be noted there is a natural correspondence between models and modules. Given any implementation M, we say that M suggests a model M' with the following domains: D_T consists of all possible compositions of procedure calls of M; D_L consists of those elements of D_T that do not result in error; D_V consists of those elements of D_T that end in a function call; and D_e is the null prodecure call. D is $D_T, D_1, \ldots, D_n$ where D_i is the ith parameter domain of the module under an appropriate ordering. If M' models a specification S when I is a function that takes a string in S to its namesake in M', then we say S specifies M.

It should be clear that every implementable specification has a model that can be constructed in this fashion. The converse is false unless the specification has a model that contains only computable functions.[4] I will say more on this later, but it should be noted that the language can be restricted so as to eliminate such functions. I have chosen not to do so here for the sake of a more elegant theory and to emphasize the distinction, often blurred in the literature, between a model and an implementation. All specifications contained in this paper are implementable.

TRACE DEDUCTIVE SYSTEM

An alternative to the semantic conception of consistency for a trace specification discussed in the previous section is that one cannot derive V(T)=d, V(T)=d', and -d=d' from the specification for any variable-free trace expression T and domain constants d and d'. An alternative to the semantic conception of sufficient-completeness for a specification is that whenever L(T.C) is derivable from the specification for any variable-free trace expression T and variable-free function call C, then one can derive V(T.C)=d for some domain constant d. In the next two sections 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 [10]. A deduction theorem for the system is proven, and possible extensions and modifications to the system 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; P any procedure call; and $\underline{A}$, $\underline{B}$, and $\underline{C}$ any assertions. $\underline{A}$t/s is the result of replacing every free occurrence of t in $\underline{A}$ by s except where such a substitution would result in a bound occurrence for s.[5]

Axioms

Any well-formed instance of the following schemata is an axiom.

1. (Ev)v=t, where v and t are the same type, t not of the form V(T) for any trace expression T nor of the form $f(t_1,\ldots,t_n)$.
2. (v)($\underline{A}$ --> $\underline{B}$) --> ($\underline{A}$ --> (v)$\underline{B}$), where v is not free in $\underline{A}$.
3. ((v)$\underline{A}$ & (Ev)v=t) --> $\underline{A}$v/t, where $\underline{A}$v/t is well-formed.
4. (Ev)v=t --> t=t
5. t=t' --> ($\underline{A}$ <-> $\underline{A}$'), where $\underline{A}$' is like $\underline{A}$ except for possibly having some occurrences of t' where $\underline{A}$ has t.
6. t=t' --> $(Ev_{d1})v_{d1}$=t [v...v $(Ev_{dn})v_{dn}$=t] for each parameter domain di such that v_{di}=t is well-formed.
7. T=T.e
8. T=e.T

9. $T \equiv R \leftrightarrow$

$$(S)((L(T.S) \leftrightarrow L(R.S)) \;\&$$
$$(-S=e \rightarrow$$
$$(((Ed)V(T.S)=d \leftrightarrow (Ed)V(R.S)=d) \;\&$$
$$((Ed)V(T.S)=d \rightarrow V(T.S)=V(R.S)))))$$

10. $L(e)$

11. $L(T.S) \rightarrow L(T)$

12. $F(D,\underline{C})$, where D is any domain name and $\underline{C}$ is a function call of type D.

13. $-F(D,\underline{C})$, where $\underline{C}$ is any procedure call that is not a function call of type D.

14. $(F(D,T.S) \;\&\; -S=e) \leftrightarrow F(D,S)$

15. $(F(D,T) \;\&\; L(T)) \leftrightarrow (Ev)V(T)=v$ where v is of type D

16. $-T=e \rightarrow$

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

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

(U. G.) Universal Generalization: if $\underline{A}$ appears as an earlier line in a derivation, then one may enter $(v)\underline{A}$ as a line in the derivation.

(E. I.) Existential Interchange: if $\underline{A}$ appears as an earlier line in a derivation and $\underline{B}$ is like $\underline{A}$ 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, (2) an axiom, or (3) an assertion justified by a rule of inference with the restriction that U. G. is not applied to any variable that occurs free in the semantic specification of S. (It should be noted that the restriction on U. G. is otiose when S is proper.).

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

Deduction Theorem

The following theorem will prove useful in later sections of this paper.

Theorem: If S is a (possibly nonproper) trace specification whose semantic specification contains the assertion B, then $S \vdash A$ only if $S' \vdash (B \rightarrow A)$ where $S' = S - \{B\}$, i. e. S with B removed from its semantic specification.

Proof: Proof is by induction on the length of derivations. As such, we will assume the theorem for all derivations of length less than n and show that it must hold for all derivations of length n whose final line is, say, A. There are four possible cases:

(1) If A is an axiom, then trivially $S' \vdash A$, from which we can infer that $S' \vdash (B \rightarrow A)$ by T.

(2) If A was inferred by T, then it is a tautological consequence of earlier lines $L_1, \ldots, L_m$. By hypothesis, $S' \vdash (B \rightarrow L_1), \ldots, S' \vdash (B \rightarrow L_m)$, from which we may infer that $S' \vdash (B \rightarrow A)$ by T.

(3) If A was inferred by U. G., then it is of the form (v)L where L appears as an earlier line. By hypothesis, $S' \vdash (B \rightarrow L)$. Now since we could apply U. G. to L, it must be the case that the generalized variable does not occur free in S. Therefore, we can apply U. G. to obtain $S' \vdash (v)(B \rightarrow L)$. Further, since B is in S, it can't be the case that v appears free in B, so we can apply axiom (2) and T to obtain $S' \vdash (B \rightarrow (v)L)$.

(4) If A was inferred by E. I. from an earlier line L, then $S' \vdash (B \rightarrow L)$, and we can apply E. I. to obtain $S' \vdash (B \rightarrow A)$.

Possible Extensions and Modifications

The above system is minimal in that although I think it correctly represents the trace 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 trace 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 [5]) of return values. Nevertheless, the various ways of implementing these extensions merit discussion.

Axiom (9) requires that if $T \equiv R$, 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 (9) 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 [3] or an intensional operator [8].

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 (16) 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 set of assertions S only if it is a semantic consequence of S, i. e., $S \vdash A \Longrightarrow S \models A$.

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 $S \vdash_m A \Longrightarrow S \models A$, 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 $S \vdash_n A \Longrightarrow S \models A$. If A is an assertion contained in S, the proof is trivial since, by assumption, M models S. If A 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.

(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) nor of the form f(argument list). 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 t=t is true iff t is a denoting term.

(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 from axiom (6), then it is of the form t=t' --> $(Ev_{d1})v_{d1}$=t [v...v $(Ev_{dn})v_{dn}$=t]. Its truth follows from the fact that t=t' can be true only if t denotes an object in some domain of the appropriate type.

(7) If A was inferred by axiom (7) or (8), 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.

(8) If A was inferred by axiom (9), then 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] $\in D_L$ if and only if I[R.S] $\in D_L$, 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.

(9) If A was inferred by axiom (10), then it is of the form L(e). But for any model this is true since I[e] is contained in I[L].

(10) If A was inferred by axiom (11), then it is of the form L(T.S) —> L(T). This is true on all models by the definition of I[.].

(11) If A was inferred by one of axioms (12)-(13), then its truth follows from the the definitions of I[procedure call] and I[F]

(12) If A was inferred by axiom (14), then it is of the form (F(D,T.S) & -S=e) <—> F(D,S). Its truth can be seen by noting that $D_V \cap$ I[e] = $\emptyset$, and that if S $\neq$ e then I[T.S] $\in D_{Vi}$ iff S $\in D_{Vi}$.

(13) If A was inferred by axiom (15), then it is of the form (F(D,T) & L(T)) <—> (Ev)V(T)=v. Its truth follows from the fact that I[V] is defined on all and only legal traces in D_V and takes elements of D_{Vi} to D_i.

(14) If A was inferred by axiom (16), 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 name 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 u $\in D_T$ and w $\in$ 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 for any assertion P, P & -P is derivable from the specification. By the Soundness Theorem, this latter 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 variable-free trace expression 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 $I[a] \neq I[b]$.

The importance of the Soundness Theorem is hard to over emphasize. All too often researchers in software specification talk of proving that one cannot derive a contradiction from a specification by giving a structure that "satisfies" the specification. Such talk is empty unless one precisely defines a deductive system and shows that the structure satisfies, not only the specification, but the deductive system as well, i. e., that the structure preserves truth under derivation. It is not generally the case that giving a structure that "satisfies" (in some intuitive sense) the assertions of a trace specification is a valid method for proving the specification syntactically consistent. One must also show that the structure preserves truth under the trace derivation system. What the Soundness Theorem demonstrates is that any structure that is a model in our technical sense satisfies the trace deductive system. Hence, to prove syntactic consistency, we must merely show that the specification's assertions are true in some model. We can totally ignore the deductive system. Analagous remarks apply to the Completeness Theorem presented in the next section.

COMPLETENESS THEOREM

The first corollary of the Soundness Theorem offers us a method for proving specifications consistent in either sense of the term, viz. finding a model, and the second corollary offers a method for proving sufficient-incompleteness, finding distinct models. In this section we will prove the deductive system complete vis-a-vis the semantics, i. e., that every syntactically consistent specification is satisfiable. This will demonstrate the universality of model construction as a method of proving specifications, e. g., consistent and will complete the bridge, began in the previous section, between the syntactic conceptions of consistency and sufficient-completeness on one hand and their semantic counterparts on the other. The reader should be warned that for the rest of this section I will use the term "trace specification" to refer to both proper and non-proper specifications, unless explicitly noted otherwise. Further, I will often refer to the semantic specification of some trace specification S, simply as S for the sake of brevity.

Theorem: Every syntactically consistent trace specification is satisfiable.

Proof: We will follow a method analagous to one employed in [7] which consists of demonstrating that every specification of a certain type is satisfiable and then demonstrating that every consistent specification can be extended to a specification of that type. The algebraically inclined reader will recognize the type of specification we will focus on as being similar to an ultrafilter. This renders the second demonstration analagous to the proof that every set of formulas in a Boolean algebra that satisfies the finite intersection property can be extended to an ultrafilter, and suggests the following definitions:

Definition: A trace specification is maximally consistent if and only if it is syntactically consistent and is such that the addition of any further assertion to its semantic specification would render it syntactically inconsistent.

Definition: A trace specification is w-complete if and only if for each assertion of the form (Ex)A in its semantic specification for some variable x, there is an assertion of the form Ax/t in its semantic specification for some term t of the same type of x, but not of the form V(T) or $f(t_1,\ldots,t_n)$.

Fact: For any maximally consistent trace specification S and any assertion A, either A is in the semantic specification of S or -A is in the semantic specification of S.

Proof: If neither A nor -A is in S, then $S \cup \{A\} \vdash (P \& -P)$ and $S \cup \{-A\} \vdash (P \& -P)$ for any closed assertion P. But then $S \vdash (A \rightarrow P \& -P)$ and $S \vdash (-A \rightarrow P \& -P)$ by the Deduction Theorem since P & -P is closed. But by T, this implies that S is inconsistent.

Corollary: S is closed under derivation, i. e., if $S \vdash A$ then $A \in S$.

Proof: If $A \notin S$ then $-A \in S$, and S would be inconsistent.

Lemma: Every maximally consistent, w-complete specification S is satisfiable.

Proof: Order all the domain names $d_1 \ldots d_m$ such that d_1-d_n name all the return value domains. Divide the terms of S not of the form V(T) nor $f(t_1,\ldots,t_n)$ into equivalence classes $E_t = \{s: s=t \text{ is in } S\}$. (These are equivalence classes, given the corrollary proven above, since reflexivity, transitivity, and symmetry are all provable for identity for such terms within the deductive system.) Since there are at most a countable number of terms, we can associate each equivalence class E_t with an unique integer N_t. In each case this integer is the denotation of every term in the equivalence class associated with it. Since there are no well-formed assertions of the form $t_1=t_2$ where t_1 and t_2 are of different types, each class contains terms of one type. Further each trace expression, domain constant, and domain variable receives a denotation since t=t is in S for each such t by axioms (1) and (4).

This assignment suggest the following domains.

$D_T = \{x: x = N_T \text{ for some trace expression } T\}$.

$D_i = \{x: x = N_t \text{ for domain variable or constant } t \text{ of type } d_i\}$.

$D_e = \{N_e\}$.

$D_L = \{x: x = N_{string} \text{ and L(string) is in the specification}\}$.

$D_{Vi} = \{x: x = N_T \text{ and } F(d_i,T) \text{ is in the specification}\}$.

The rest of the specification receives the following assignment.

$I[B] = N_B$ where B is a domain variable or constant.

$I[f]$ = the function that takes $(N_{t1},\ldots,N_{tn})$ to N_a such that

$f(t1,\ldots,tn)=a$ is in S.

$I[R] = \{(N_{t1},\ldots,N_{tn}): R(t1,\ldots,tn) \text{ is in } S\}$.

$I[e] = N_e$.

I[procedure PROC that takes n parameters] = the function that takes

$(N_{P_1},\ldots,N_{P_n})$ to $N_{PROC(P_1,\ldots,P_n)}$ where

$P_1,\ldots,P_n$ are appropriate parameters for PROC.

$I[F] = \{\{D_i\} \times D_{Vi}: 1 \leq i \leq n\}$.

$I[L] = D_L$.

$I[=] = \{(x,x): x = N_t \text{ for some } t\}$.

$I[.]$ = the function that takes (N_T,N_S) to $N_{T.S}$.

$I[V]$ = the function that takes any integer N_T in D_V D_L to the

integer N_a such that $V(T)=a$ is in the specification.

$I[\equiv] = \{(x,y): x = N_T \text{ and } y = N_S \text{ and } T\equiv S \text{ is in the specification}\}$.

The following considerations show that the domains specified meet the conditions for being a model. The fact that x is in D_T implies that $x \in D_e \cup \{y: y = I[.](u,w) \text{ for some } u \in D_T \text{ and } w \in Rng(Rng(I/\{v: v \text{ is an access procedure}\}))\}$ follows from the fact that S is consistent, is w-complete, and contains axiom (16) since S is closed under derivation. If N_T is not in D_e, then $T=e$ is not in S, and hence, given the fact that for every A either A or $-A$ is in S, $-T=e$ is in S. But this implies that the consequent of axiom (16) is in S. Since S is w-complete, an instance of this consequent is also in S, and since the union of this disjunction with the set containing the negation of each disjunct is inconsistent, one of the disjuncts must be in S. Hence, we have an assertion of the form T=R.C in S where C is a procedure call. $D_e \cap D_V = \emptyset$ since $F(d_i,T) \dashrightarrow -T = e$ is in S via axiom (14), and $D_e \subset D_L$ by axiom (10). Every trace in D_V is in some D_{Vi} by axioms (12) and (16), and no trace is in two D_{Vi} by axioms (13) and (16).

The fact the clauses (1), (2), (3), (7)-(10) of the definition of trace sequence are satisfied follows by construction of the sequence given that S is a well-formed trace specification. The fact that the rest of the definition is satisfied rests on the fact that S is maximally consistent, w-complete and closed under derivation.

With respect to clause (4), note that if N_T is in $D_{Vi} \cap D_L$, then T is in an equivalence class that contains some trace expression R and some trace expression Q such that $F(d_i,R)$ and L(Q) are in S. But this can be the case only if T=R and T=Q are in S, which given axiom (5) and the fact that S is closed under derivation, implies that $F(d_i,T)$ and L(T) are in S. This implies that $(Ev_{di})V(T)=v_{di}$ is in S by axiom (15), which in turn implies that V(T)=t is in S for some domain variable or constant t of type d_i since S is w-complete. Hence, I[V] takes T to the domain which is associated with t, viz. d_i.

Case (a) of clause (5) is trivial and cases (b) and (c) follow from axioms (7) and (8). Case (d) follows from the fact that if $N_T \notin D_V \cup D_e$ then for all d_i, $-F(d_i,T)$ and -T=e must be in S. But then by axiom (14), there can be no R such that $F(d_i,R.T)$ is in S. Clause (e) follows from the fact that if $N_T \in D_{Vi}$ then $F(d_i,T)$ must be in S as shown in the preceding paragraph. But then for every R, $F(d_i,R.T)$ is in S by axiom (14), which implies that $N_{R.T} \in D_{Vi}$ for every R. Finally, case (f) follows from axiom (11) by an analagous argument.

Clause (6) is satisfied by a similar argument in light of axiom (9). Case (a) and case (b) of clause (11) are trivial, and case (c) is satisfied by standard argument given axioms (12) and (13).

Given that we are dealing with a sequence, we must now show that it satisfies every assertion in S. To this end, define the order of an assertion A, O[A], as follows:

If A is of the form R(t1,...,tn), L(T), F(T), T≡R, or t=t' then O[A] = 1.

If A is of the form B v C, B & C, B —> C, or B <—> C then

O[A] = max(O[B],O[C]) + 1.

If A is of the form (v)B or (Ev)B then O[A] = O[B] + 1.

We can now show that every assertion A in S is satisfied by induction on the order of A. Assuming that each formula of order less than n is satisfied iff it is in S we will show that each formula A of order n is satisfied iff it is in S. There are four possible cases:

(1) If O[A] = 1 and A is not of the form t=t', then the sequence satisfies A iff A is in S by construction. If A is of the form t=t', we must show that t (and hence, by symmetry, t') denotes something. There are two cases. For t not of the form V(T), if t=t' is in S then (Ev)t=v is in S

by axiom (6). For t of the form V(T), we have $(Ev_{d1})V(T)=v_{d1}$ v...v $(Ev_{dn})V(T)=v_{dn}$ is in S by axiom (6). But as shown in establishing the validity of our domains above, a disjunction can be in S only if a disjunct is in S. Hence, (Ev)V(T)=v is in S for some v_{di}. This implies that if t=t' is in S, then t denotes something. Given this fact, A is satisfied by construction.

(2) If O[A] is greater than 1 and A is a truth functional compound of B and C, then by induction hypothesis, B and C are in S iff they are satisfied. Consider the case where A is of the form B v C. If A is in S, and neither B nor C are in S, then -B and -C are in S by familiar argument, and S would be inconsistent. Therefore, if A is in S either B is in S or C is in S. Hence, if A is in S, either B or C is satisfied, and therefore, A is satisfied. If A is not in S, then -A is in S. Hence, neither B nor C can be in S since S is consistent. Hence, neither B nor C is satisfied, from which it follows that A isn't satisfied. The argument for other truth functional compounds is similar.

(3) If O[A] is greater than 1 and A is of the form (Ev)B, then A is in S only if Bv/t is in S for some term t not of the form V(T) or f(t1,...,tn) since S is w-complete. Now Bv/t is satisfied by induction hypothesis which implies that B is satisfied by that sequence which is identical to the one we constructed except for assigning N_t to E_v. Hence, A is satisfied by definition. If A is not in S then -A is in S, and there can be no t such that Bv/t is in S since S is consistent. Hence, by induction hypothesis, there is no sequence of the appropriate type that satisfies B since by construction, every element in D is denoted by some t. Therefore, A is not satisfied.

(4) The only other possibility is if A is of the form (v)B. If A is in S, then Bv/t must be in S for every term (not of the form V(T) or f(t1,...,tn)) of the same type as v by familiar argument. Hence, by induction hypothesis, B is satisfied by all appropriate sequences since every element in D is denoted by some t. Therefore, A is satisfied. If A is not in S, then -(v)B must be in S by familiar argument. Hence, (Ev)-B is in S since S is closed under derivation. But since S is w-complete, this implies that -Bv/t is in S for some t. Hence, there is a sequence which fails to satisfy B, and therefore, A is not satisfied.

Given that we have established that every maximally consistent, w-complete specification has a model, all that is left to prove is that any consistent trace specification is contained in some maximally consistent, w-complete specification.

Lemma: Every syntactically consistent specification can be extended to a maximally consistent, w-complete specification.

Proof: Enumerate all assertions in the trace specification language so that A_i is the ith assertion in the enumeration and construct the set S as follows:

S_0 = the original set.

$S_{i+1} = S_i$ if $S_i \cup \{A_{i+1}\}$ is syntactically inconsistent

$S_{i+1} = S_i \cup \{A_{i+1}\}$ if the resulting set is syntactically consistent and A_i is not of the form -(v)B.

$S_{i+1} = S_i \cup \{A_{i+1}, -Bv/t\}$ where t is the alphabetically first variable the same type as v that does not appear in $S_i \cup \{A_i\}$, if

$S_i \cup \{A_{i+1}\}$ is syntactically consistent and A_i is of the form $-(v)B$.

$S = S_w$.

The fact that S is maximal can be seen by noting that if some assertion A_{i+1} is not in S, then it is because $\{A_{i+1}\} \cup S_i$ was inconsistent. But for any assertion A, if $\{A\} \cup S_i$ is inconsistent, then $\{A\} \cup S$ must be inconsistent since S_i is a subset of S.

The fact that S is syntactically consistent can be established if we demonstrate that each S_i is syntactically consistent since any derivation of an inconsistency can involve at most a finite number of premises and every finite set of assertions contained in S is contained in some S_i. We can establish the syntactic consistency of each S_i by induction. S_0 is consistent by hypothesis. Now, if S_i is consistent, then S_{i+1} is obviously consistent if either it is equal to S_i or it was formed by the addition of some formula which could be added consistently to S_i. Hence, the only problematic case occurs when we add $-(v)B$ to S_i since in this case we also add a formula of the form $-Bv/t$, and we have no guarantee that this latter formula is consistent with the resulting set. However if $S_i \cup \{-(v)B, -Bv/t\}$ is inconsistent, then we can derive P & -P from this set for some closed asserton P. But by the Deduction Theorem, this implies that we can derive Bv/t from $S_i \cup \{-(v)B\}$ by using the T rule of inference. Since by hypothesis, t does not occur free in S_i or in $-(v)B$, we can generalize and derive $(t)B$ which implies that $S_i \cup \{-(v)B\}$ could not have been consistent.

The fact that S is w-complete can be seen by noting that if (Ev)A is in S then -(v)-A must be in S. But when this latter formula was added, we also added the formula --Aa/t for some variable t. But this formula can be in S only if Aa/t is also in S.

<u>Corollary</u>: Every syntactically consistent proper specification has a model.

Proof: Immediate given that a set of closed assertions is satisfiable by a sequence iff the set is true in every model that contains that sequence.

<u>Corollary</u>: $S \models A$ if and only if $S \vdash A$.

Proof: The implication from right to left follows from the Soundness Theorem. Going from left to right, note that if A is not derivable from S, then S U $\{-A\}$ is syntactically consistent. Hence, by the Completeness Theorem it is satisfiable by some sequence, and therefore, it can't be the case that $S \models A$.

<u>Corollary</u>: S is syntactically consistent (sufficiently-complete) iff it is semantically consistent (sufficiently-complete).

Proof: Immediate given the preceding corollary.

APPLICATIONS

We have seen a syntactic and a semantic definition of consistency and of sufficient-completeness, and we have seen that the syntactic and semantic definitions are coextensive in each case. Although the fact that the definitions are coextensive provides independent evidence that each adequately captures what we are after, some may feel that the coextensiveness demonstrates that we only needed, e. g., the syntactic definition to begin with. However, there are advantages in having two definitions which come to light when considering proofs about specifications. These advantages stem from the fact that ceteris paribus, it is easier to prove that something exists with a certain property (e. g., that there are koala bears in Australia) than to prove that nothing exists with a certain property (e. g., that there are no polar bears in Australia). After all, the method of proof in the first case is obvious and indisputable: show the beast. It is worthwhile considering specific examples.

Application to Consistency

When demonstrating that a specification is inconsistent, it is usually easier to directly derive P & -P from the specification for some assertion P, than to directly demonstrate that the specification has no model. However, when demonstrating that a specification is consistent, it is usually easier to provide a model than to directly demontrate that P & -P cannot be derived. As an example, a direct syntactic demonstration that P & -P cannot be derived from the following stack specification when supplemented by first order number theory is quite difficult, while it can easily be shown that the supplemented specification has a model.

Stack Specification:

a and b are assumed to be a type integer, while r and s are assumed to be of type name. $\pm$ is written in infix, and subscripts are dropped.

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) $-r=s \rightarrow$ T.PUSH(a,s).PUSH(b,r)$\equiv$T.PUSH(b,r).PUSH(a,s)

(6) L(T.TOP(s)) $\rightarrow$ T.TOP(s)$\equiv$T

(7) L(T) $\rightarrow$ V(T.PUSH(a,s).TOP(s))=a

(8) L(T) $\rightarrow$ V(T.PUSH(a,s).DEPTH(s))=V(T.DEPTH(s))+1

(9) (L(T) & $-r=s$) $\rightarrow$ 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 = (D_T,int,name) where $D_T = \{x$: x is a possibly empty, variable-free string of procedure calls$\}$. $D_V = D_{V1}$ = those strings that end in TOP or DEPTH, D_e is the empty string, and $D_L = \{x$: x is in D_T and is 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 assigns to L, F, e, and = the obvious intepretation and to each numerical constant and function the standard interpretation.

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[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[≡] = $\{$(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.

An important aspect of the above proof is that it demonstrates, not merely that the stack specification has a model, but also that it has a model of the type we are interested in. For example, if we wrongly assumed that DEPTH returned, not the present depth of the stack, but rather the maximum depth that the stack had attained in its history, we would discover that we could find no model that interpreted DEPTH in the desired way. If we had proven the specification consistent by syntactic means, however, this fact would have never come to light.

Application to Sufficient Completeness

A direct syntactic proof that a specification is not sufficiently-complete would consist in demonstrating that there is a variable-free trace expression

T ending in a function call such that L(T) is derivable but V(T)=$\underline{a}$ is not derivable for any constant $\underline{a}$. Such a proof would be, at best, challenging. However, the semantic approach is straight forward. One must merely give two models M and M' for the specification and show that V(T)=$\underline{a}$ is true in M and false in M', thus demonstrating that there can be no $\underline{a}$ such that V(T)=$\underline{a}$ is derivable. As an example, I will prove the following keysort specifications, adapted from a similar one suggested by David Parnas, incomplete[6]:

Keysort Specification:

It is assumed that a, a', b, b', x, and y are of type integer. $\geq$ and $\leq$ are written in infix, and subscripts are dropped.

Syntax:

INSERT: (int) x (int)

REMOVE:

FRONT: --> (pair)

int = the set of integers

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

Semantics[7]:

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

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

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

```
    (4)  V(T.INSERT(a,b).FRONT)=(a,b) -->
              (T.INSERT(a,b).REMOVE ≡ 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 ≡ 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 and M' agree on the following:

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

(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$\}$.

For M 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 string.

3. Label the ith INSERT from the left and the jth REMOVE from the left in string 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 string, 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 string.

   (b) If INSERT_i is to the left of REMOVE_k, then $\text{key}_i \geqslant \text{key}_j$.

   (c) $i>j \Longrightarrow$ (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) $I[\Xi] = \{(x,y)$: $x,y \in D_T$ 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 $\text{INSERT}_i < \text{INSERT}_k$ if $\text{key}_i < \text{key}_j$ or if $\text{key}_i = \text{key}_j$ & $i < j\}$.

For M', 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 string, 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) $i > j \Longrightarrow$ (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 and V(A) = (1,5) in M'.

With respect to proving a specification sufficiently-complete, there is a standard semantic approach that suggests itself. Call a specification _theory complete_ if for every assertion A in the specification language, either A or -A is derivable. If we can show that a specification is theory complete and give one model in which for every legal, variable-free trace expression T ending in a function call, V(T)=_a_ is true for some constant _a_, then it follows that the specification is sufficiently-complete. What is appealing about this approach is that there are standard methods for proving theory completeness [2]. However, the approach is very limited in that any specification containing first order number theory is not theory complete [3]. Further, I suspect that even specifications that do not appeal to first order number theory are not theory complete for reasons to be discussed in the section of this paper on future research.

Nevertheless, there are two relatively straight forward approaches to proving a specifcation sufficiently-complete, neither of which seems preferable to the other. One is to show by induction on the length of a trace expression that for any variable-free trace expression T there is a constant $\underline{a}$ such that $V(T)=\underline{a}$ is derivable. The other is give a model that makes $V(T)=\underline{a}$ true for all appropriate T and then perform an induction analagous to the one described in the first approach to demonstrate that every other model must also make $V(T)=\underline{a}$ true. As an example, I will use to first approach to prove the keysort specification sufficiently-complete when we replace assertion (7) of the specification by the following:

```
(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')))
```

Assume that for all variable free strings of procedure calls T ending in FRONT of length less than n such that L(T) is derivable there is some ordered pair of integers $\underline{a}$ such that $V(T)=\underline{a}$ is derivable. We must show that for all such strings T of length n that there is such an $\underline{a}$.

First note that by the soundness theorem, if T legal, i. e., L(T) is a theorem, then L(T) is true in all models. Therefore, every legal string must be of the form described in M (although it is not necessarily the case that for every T such that L(T) is true in M, T is legal). T can be of four possible forms:

(1) If T is simply the call FRONT, then it is not the case that L(T).

(2) If T is of the form S.FRONT.FRONT, then V(T)=V(S.FRONT) by assertion (3), and by the induction hypothesis, there is an $\underline{a}$ such that $V(S.FRONT)=\underline{a}$ is derivable.

(3) If T is of the form S.INSERT.FRONT and S.FRONT is not legal, then we can use assertions (3), (4) and (5) to derive S e. We can then use assertion (6) to derive a value for T. If S.FRONT is legal, then by the induction hypothesis V(S.FRONT) has a value, and we can use this value with assertion (7') to derive a value for V(T).

(4) If T is of the form S.REMOVE.FRONT, we know that V(S.FRONT) has a value. We can use this value with assertions (3), (4), and (5) to prove that T is equivalent to a shorter expression and apply the induction hypothesis.

COMPARISON WITH THE ALGEBRAIC APPROACH

The trace method has much in common with the more algebraic approaches to specification, as epitomized, for example, by Guttag and Horning [4]. They are both methods of "abstract" specification and therefore, seem very much alike when compared to such alternatives as the "operational definition" and "abstract model" approaches discussed earlier. Further, the generality of the term "algebraic" allows the development of algebraic models that are formally equivalent to the trace method.[8]

Nevertheless, the reader will notice two differences between the trace method for specifying software modules and the algebraic approach. First, 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 regard each procedure as taking "stack" as a parameter and any procedure that affects the "inner state" of the module (called O-functions in [1]) as returning a stack. There is certainly no reason for the procedures in an implementation of the module to contain such an abundance of parameters and return values, yet if the user is given the freedom to leave certain parameters out of his implementation, we lose the advantages of abstract specification discussed earlier. But how are these parameters to be represented? Few programming languages allow for the free creation of new data types. Hence, the interface is ambiguous in that the programmer must decide whether to treat the parameter as a name or as one of several possible data objects, e. g., an array. Choosing to represent the parameter as a data object would be particularly bad since it would force the programmer to represent each stack as a separate data object, ruling out implementations that use, e. g., only one array that stored both names and integers. The programmer faces similar problems in dealing with the artificial error values the algebraic approach needs for its stacks to assume and the unnecessary functions it needs in order to start, i. e., map an empty value space to an initial stack.

A slightly different problem that results from treating stacks as values is that it obliterates the distinction between a function call and the value returned by that call. This renders it impossible to represent a sequence such as $call_1.call_2$ except by treating $call_1$ as a parameter of $call_2$.[9] This is not only unintuitive for many implementations, it also makes it impossible to represent sequences such as PUSH(i,s).TOP(s).TOP(s) since the first occurrence of TOP returns an integer while the second occurrence needs a stack for a parameter.

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 (14) 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 methods employed in this paper constitute an important step toward coming up with a uniform method for establishing trace specifications consistent and sufficiently-complete. The next step is described as an area for future research.

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, alternative methods for proving specifications consistent and sufficiently-complete should be studied. One possibility is to cast the specification language as a reduction language and use methods suggested in [4]. Another is to formalize a specification and the trace deductive system in first order number theory, as in [3], and then try to derive a formula that says intuitively that the specification is consistent or sufficiently-complete. Such an approach depends on finding appropriate bounds, e. g., on the length of of a derivation proving an expression legal given the length of the expression. Both methods can be computerized, given a sufficiently efficient theorem prover. Such a theorem prover could also be employed in generating implementations from specifications by keeping track of procedure calls and deriving $V(T)=\underline{a}$ when appropriate. Naturally, such implementations cannot be found for specifications of noncomputable functions, but these can be eliminated by restricting the specification language, e. g., by bounding quantification, or by placing restrictions on what can count as a specification, e. g., by making it mandatory that certain assertions are provable. A pilot project to develop software support was undertaken at the University of North Carolina and is being continued at the Naval Research

Laboratory. Third, methods for proving the correctness of implementations and the correctness of programs using modules must be developed. 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 axiomatically force 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. This lack of expressive power stems from the fact that first order logic is compact [2]. Issues concerning practical consequences of this fact should be explored. One such consequence may prove to be that trace specifications, in general, are theory incomplete. Since higher order logics and set theory are not compact, they have more expressive power. However, they are also not complete. Although noncompact languages must be strongly incomplete if only finite derivations are allowed, future research should explore the possibility of noncompact languages that are weakly complete [2]. Such languages should have the expressive power to rule out the nonstandard models mentioned above.

ACKNOWLEDGMENTS

The influence of David Parnas' work on this paper is obvious. I also wish to thank Karen Dwyer, Mila Majster, Donald Stanat, and David Weiss for helpful criticism of and earlier draft, and Mark Nixon for some very useful conversation along the way. Part of the research for this paper was done under NSF grant #1-0-110-3276-XA452.

FOOTNOTES

1. The predicate F, which intuitively holds of a trace expression if and only if that expresion ends in a function call, was not included as part of the original language in [1]. However, it is necessary if we are to have a natural trace deductive system that is complete with respect to a natural semantics. The inclusion of F is only one of many practical consequences resulting from work in the area of theoretical foundations of the trace method.

2. The following formation rules are given in Backus Normal Form.

3. Ideally, D_T is the smallest set closed under composition that contains the null trace and each procedure call. However, this restriction cannot be forced axiomatically in first order logic. See the future research section of this paper.

4. As a counterexample, the interested reader can verify that Gödel's "provable formula" predicate [3] is specifiable though not recursive.

5. The notion of occurrence used here is the standard one as used, e. g., in [10] extended so as to regard trace expressions that occur within other trace expressions as occurring in any assertion that the latter occurs in.

6. 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.

7. 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.

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

9. On the other side of the coin, it should be noted that the trace method makes a sharper distinction between functi›n 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.

BIBLIOGRAPHY

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

2. Chang, C. and Keisler, H. Model Theory, 2nd ed. (Amsterdam 1977).

3. Gödel, K. "Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme, I", Monatshefte für Mathematik und Physik, XXXVIII (1931), pp. 179-98.

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

5. Hehner, E. Simple Set Theory for Computing Science, CSRG Technical Report #102 (1979)

6. Heitmeyer, C. and McLean, J. "An Approach to Describing the Functional Requirements of an Embedded Communications System", forthcoming.

7. Henkin, L. "The Completeness of First Order Functional Calculus", Journal of Symbolic Logic, XIV (1949), pp. 159-166.

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

9. Liskov, B. and Berzins, V. "An Appraisal of Program Specifications", Research Directions in Software Technology, ed. P. Wegner (Cambridge, Massachusetts 1979).

10. Mates, B. Elementary Logic, 2nd ed. (New York 1972).

11. Monk, J. Introduction to Set Theory (New York 1969).

Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, XIV (1972), pp. 1053-1058.

13. _______. "The Use of Precise Specifications in the Development of Software", Information Processing 77, ed. B. Gilchrist (New York 1977).

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