Trace Specifications: Methodology and Models

DANIEL HOFFMAN, MEMBER, IEEE, AND RICHARD SNODGRASS

Abstract—Precise abstract software specification is achievable by using formal specification languages. However, nontrivial specifications are inordinately difficult to read and write. This paper summarizes the trace specification language and presents the trace specification methodology: a set of heuristics designed to make the reading and writing of complex specifications manageable. Also described is a technique for constructing formal, executable models from specifications written using the methodology. These models are useful as proofs of specification consistency and as executable prototypes. Fully worked examples of the methodology and the model building technique are included.

Index Terms—Formal specification, logic, prototype, software engineering.

I. Specification Issues

In this paper, we present a set of heuristics that aid in the writing of good specifications of complex modules. By *specification* we mean a precise description of the essential behavior of a software module. We wish to concentrate on the correctness of the software and so we do not include speed or cost requirements in our specifications. The specifications are expressed in the trace language, developed by Parnas and Bartussek and formalized by McLean [2], [18]. In the remainder of this section, we discuss the role of specifications in software development.

We envision our specification technique as a designer's tool, as discussed by Parnas [20], [21]. Nearly every software design methodology is based on the decomposition of large, complex modules into smaller, simpler ones. Then each module can be dealt with separately, often handled by different people or teams. Each module can be further decomposed, or, if it is simple enough, implemented directly. If the modules are to cooperate successfully at system integration time, then their specifications must be clearly understood. A module *M*'s specification must be understood by both the implementors of *M* and the implementors of every module that uses *M*. Thus, it is crucial to precisely and completely record the specifications of each module.

Specifications can also support the design process through design verification. Testing is used to detect design errors as they appear in an implementation. We would like to detect design errors *before* the design is implemented, and avoid the cost of changes to the implementation resulting from late detection of design errors. Without precise specifications it is difficult to verify the correctness of a decomposition. *Formal* specifications, written in a language with a formal syntax and semantics, have two important advantages over informal specifications. First, formal specifications avoid the ambiguity and imprecision inherent in prose, and second, formal specifications are machine processable, crucial for specification verification. *Abstract* specifications are written solely in terms of the observable behavior of the module. In contrast, *operational* specifications are written in terms of a program, usually written in some procedural language, that performs the desired task [1]. We favor abstract specifications because they focus attention on the requirements of each module (its specification) as opposed to the method used to fulfill those requirements (its implementation). The advantages of abstract specifications have been discussed in more detail elsewhere [2], [9], [18].

The remainder of this paper is divided into five sections. Section II describes the trace language and Section III the trace specification methodology. Section IV defines trace models and illustrates their use. Section V compares the trace and algebraic specification methods and Section VI presents our conclusions.

II. The Trace Language

In this section we describe the trace language, demonstrate it in several examples and argue the need for a specification methodology.

A. Basic Elements

Traces are a technique invented by Parnas and Bartussek for the formal and abstract specification of software modules [2]. McLean continued their work by establishing a formal basis for traces, including a formal syntax and semantics, a derivation system, and completeness and soundness results [18]. In a trace specification, a module's behavior is described in terms of procedure or function calls and return values from those calls. Following McLean's description, modules are specified by describing three properties they must possess:

1) What are the names of the module's access procedures and functions, and what are their parameter and return value types, if any? These properties are described by *syntax* sentences of the form:

$$name: _type \cdots _type \rightarrow return_value_type$$

Manuscript received April 30, 1986; revised July 31, 1986. This work was supported in part by the Natural Science and Engineering Research Council of Canada under Grant A8067.

D. Hoffman is with the Department of Computer Science, University of Victoria, Victoria, B.C. V8W 2Y2, Canada.

R. Snodgrass is with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27514.

IEEE Log Number 8822447.

0098-5589/88/0900-1243$01.00 © 1988 IEEE

2) Which series of procedure calls, termed *traces*, are legal, i.e., are not regarded as being in error? These are indicated by *semantic* assertions of the form $L(trace)$. The behavior of the module in response to illegal trace is undefined.

3) What is the output of a legal trace that ends in a function call? This value is denoted by semantic assertions of the form $V(trace) = value$.

Trace specifications also contain symbols from predicate calculus. In particular $\rightarrow$ and $\leftrightarrow$ are used for *if then* and *if and only if*, respectively. The connectives $\neg$, &, and | are used for *not*, *and*, and *or*, respectively, as well as the existential quantifier $(\exists\alpha)$ for *there exists* α, the universal quantifier $(\forall\alpha)$ for *for all* α, and the comparison operators $<, \leq, =, \neq, \geq, >$. Operator precedence is as follows:

highest $\equiv$ $<$ $\leq$ $=$ $\neq$ $\geq$ $>$ ($\equiv$ defined below)
$\neg$
& |
lowest $\rightarrow$ $\leftrightarrow$

The dot (.) concatenates procedure calls, *e* denotes the empty trace, and comments are delimited by /* and */.

In order to make specifications more readable, the following abbreviation is used. If two traces, T_1 and T_2, agree on legality and return value with respect to future module behavior, then we say they are *equivalent* and write $T_1 \equiv T_2$. More precisely, any two traces T_1 and T_2 are equivalent if

$$(\forall T)\ ((\ L(T_1.T) \leftrightarrow L(T_2.T))\ \&$$
$$((T \text{ is not empty}) \rightarrow ($$
$$(T_1.T \text{ has a value} \leftrightarrow T_2.T \text{ has a value})\ \&$$
$$((T_1.T \text{ has a value}) \rightarrow V(T_1.T) = V(T_2.T)))))$$

We also assume that any prefix of a legal trace is legal. More formally:

$$(\forall S,T)\ (L(S.T) \rightarrow L(S)).$$

B. A Stack Specification

Consider the specification of a stack module (taken from McLean [18]) that contains three procedures: *push* takes an integer parameter and returns no value; *pop* neither takes a parameter nor returns a value; and *top* takes no parameter and returns an integer value. The full specification is shown in Fig. 1.

The semantics of the module consists of five assertions describing the module's behavior:

1) If a trace has not resulted in an error, then *push* can be legally called with any integer parameter (the stack is unbounded).

2) Calling *top* will not result in an error if and only if calling *pop* does not (the stack must be nonempty).

3) Calling *push* followed by *pop* will not affect the future behavior of the module (*pop* cancels *push*).

4) If *top* can be legally called, then calling it will not affect the future behavior of the module (*top* does not affect the "internal state" of the module).

5) The value of any legal trace ending in *push* followed by *top* is the parameter of that *push*.

```
NAME
    stack

SYNTAX
    push:   integer;
    pop:    ;
    top:    → integer;

SEMANTICS
    /*1*/  (∀ T,i) (L(T) → L(T.push(i)))

    /*2*/  (∀ T) (L(T.top) ↔ L(T.pop))

    /*3*/  (∀ T,i) (T ≡ T.push(i).pop)

    /*4*/  (∀ T) (L(T.top) → T ≡ T.top)

    /*5*/  (∀ T,i) (L(T) → V(T.push(i).top) = i)
```

Fig. 1. Stack specification.

We claim that this specification captures the essence of an unbounded integer stack. For example, to find the value of a legal trace ending in *top*, we apply assertions 3 and 4 to remove all *pop* and *top* calls, respectively, and then apply assertion 5 to provide the value desired.

C. A Queue Specification

Fig. 2 shows a trace specification of an unbounded integer queue taken from Parnas and Bartussek [2]. Note that while the stack and queue syntax sections are very similar, their semantic sections are quite different. Informally, the semantics assertions state that:

(1–3) Traces which consist of any number of *add* calls are legal. A *remove* or *front* call is also permitted, if that call is preceded directly by an *add*.

(4–6) There are other legal traces. A *front* call has no effect on the future behavior of the module. The sequence *add.remove* may be replaced by *remove.add* or, if it occurs at the start of a trace, deleted.

(7, 8) These assertions show the value of *front* for a queue of length one and a queue of length greater than one.

To find the value of a legal trace containing *removes* and ending in *front*, we repeatedly apply assertion 5, shifting the *remove* left until it is beside the first *add*, then apply assertion 6, deleting the *remove-add* pair. This process is repeated until we have an equivalent trace that does not contain a *remove*. We then apply assertion 8 to determine the value returned by *front*.

D. More Complex Specifications

We have shown how traces can be used to write compact and fairly readable stack and queue specifications. Yet, when we attempted specifications of larger, more complex modules, difficulties arose. Previously developed trace specifications were short, rarely more than 25 lines long. Hence difficulties in scaling were not apparent in these specifications. Due to the complexity of the modules we were specifying, much longer specifications were necessary. The assertions in the semantics section quickly became large and unintelligible. We were often unable to determine whether a specification contained assertions that

```
NAME
    queue

SYNTAX
    add:     integer;
    remove: ;
    front:        → integer;

SEMANTICS
    /*1*/  (∀ T,i) (L(T) → L(T.add(i)))

    /*2*/  (∀ T,i) (L(T) → L(T.add(i).remove))

    /*3*/  (∀ T) (L(T.remove) ↔ L(T.front))

    /*4*/  (∀ T) (L(T.front) → T.front ≡ T)

    /*5*/  (∀ T,i) (L(T.remove) →
                T.add(i).remove ≡ T.remove.add(i))

    /*6*/  (∀ i) (add(i).remove ≡ e)

    /*7*/  (∀ i) (V(add(i).front) = i)

    /*8*/  (∀ T,i) (L(T.front) → V(T.add(i).front) = V(T.front))
```

Fig. 2. Queue specification.

were mutually contradictory or whether the specification completely characterized the behavior of the module. Small changes to the desired characteristics of a module resulted in disturbingly large changes to the specification. These experiences motivated us to develop a trace specification methodology consisting of a set of five heuristics designed to make the writing of large trace specifications manageable.

III. The Trace Specification Methodology

In this section we present the trace specification methodology and illustrate it on new stack and queue specifications, and on a more complex example, a *traversing stack* specification. The first step in developing a trace specification is writing the syntax section. Then, using the heuristics described below, we write the semantics section, transforming our informal notions of correct module behavior into assertions about trace legality, equivalence and value.

Heuristic—*Base the Specification on a Normal Form:*

A *normal form* is a representative subset of the legal traces. For a given specification S, consider the trace relation "≡", and the set T_L of all legal traces. It follows from the definition of "≡" (in Section II-A) that it is an equivalence relation and so partitions T_L into a set of equivalence classes. We require that a normal form for S contain at least one trace from each of the equivalence classes. A normal form is "representative" in the sense that, for any legal trace T, there is a normal form trace T_{NF} equivalent to T. For any specification, the set T_L is itself a normal form. Typically, specifications have an infinite number of normal forms, most of which are uninteresting. We choose a normal form which is simple to describe and which makes the next heuristic easy to apply.

Heuristic—*Structure the Semantics According to Normal Form Prefixes:*

Given a normal form, we may express module behavior for all traces in terms of module behavior for just the normal form traces. Specifically, we base the semantics assertions on traces of the form $T.C$, where T is normal form and C is a single call. Then, for each call C:

State whether $T.C$ is legal.
If $T.C$ is legal then
  If $T.C$ is not normal form then provide a normal form trace T_{NF} where $T_{NF} \equiv T.C$
  If C is a function call, specify $V(T.C)$.

Using this heuristic helps ensure that the legality and value of every trace is specified.

Heuristic—*Use Predicates:*

Using the trace language, define predicates and functions on traces. Use these to decompose a single complex assertion into simpler assertions, much as procedures are used in programming languages. Usually a predicate on traces is defined which is true for exactly those traces in the normal form.

The remaining two heuristics are not easily demonstrated on small examples and so are described only briefly here (but in detail elsewhere [13], [14]).

Heuristic—*Develop Specifications Incrementally:*

Develop a specification for a complex module as a series of specifications, beginning with a specification for a much simpler module and culminating with the full specification.

Heuristic—*Use Macros to Hide Record Structure:*

Especially in protocol specifications (see Section III-D), macros can be used effectively to hide the complex record structure of the messages passed between modules.

We present three functions, informally defined in Fig. 3, to provide information about traces and domain variables and to make certain logical constructions more convenient (see [13] for formal definitions). Individual applications may also define additional functions; those described here are of general use.

A. A New Stack Specification

We now illustrate the built-in functions and heuristics just described by respecifying the stack module (see Fig. 4). We choose the stack normal form to be the set of traces consisting solely of *push* calls. Although there are an infinite number of possible normal forms for this module, our choice is attractive because there is a natural correspondence between a sequence of *push* calls and our conception of the contents of the stack. We write the *normalform* predicate in the PREDICATES section and then

length (T)
length (T) = n if and only if n is the number of calls of any name in T. *length* (*push* (1).*pop*.*push* (2).*pop*) = 4 is true.

count (c, T)
count (c, T) = n if and only if T has n calls with name c. *count* (*pop*, *push* (1).*pop*.*push* (5).*pop*) = 2 is true.

prefix(S, T)
prefix(S, T) is true if and only if S is a prefix of T. *prefix*(*push* (1), *push* (1)) and *prefix*(*push* (1), *push* (1).*push* (2)) are both true.

Fig. 3. Builtin functions.

```
NAME
    stack

SYNTAX
    push:   integer;
    pop:    ;
    top:    → integer;

PREDICATES
    normalform(T) ↔ length(T) = count(push,T)

SEMANTICS
    ( ∀ T,C) (normalform(T) & length(C) = 1 →
        ( ∀ i) (C = push(i) →
        /*L*/  L(T.C)
        ) &
        (C = pop →
        /*L*/  (L(T.C) ↔ T ≠ e) &
        /*E*/  (L(T.C) → ( ∀ T1,i) (T = T1.push(i) → T.C ≡ T1))
        ) &
        (C = top →
        /*L*/  (L(T.C) ↔ T ≠ e) &
               (L(T.C) →
        /*E*/      T.C ≡ T &
        /*V*/      ( ∀ T1,i) (T = T1.push(i) → V(T.C) = i))
        )
    )
```

Fig. 4. New stack specification.

use this predicate to structure the SEMANTICS section. For a trace not in normal form, the equivalence clauses may be used to reduce it to normal form, by eliminating *top* calls and *push-pop* pairs. For each *push*, *pop*, and *top* call, we write assertions describing the legality (/*L*/), equivalence (/*E*/), and value (/*V*/) of a single occurrence of that call, when preceded by a normal form trace. For example, consider the *top* call. The specification states that *top* is legal exactly when the stack is nonempty, that *top* has no effect on the future behavior of the module, and that *top* returns the value most recently pushed. While this specification is longer than the stack specification shown in Fig. 1, it is just as comprehensible. Additionally, it is easier to see that the assertions are orthogonal and express all desired legality, equivalence and value constraints.

B. A New Queue Specification

In this section, we use the methodology as a tool to write a new *queue* specification, shown in Fig. 5, in the same manner as for the new *stack* specification. Note the

```
NAME
    queue

SYNTAX
    add:    integer;
    remove: ;
    front:  → integer;

PREDICATES
    normalform(T) ↔ length(T) = count(add,T)

SEMANTICS
    ( ∀ T,C) (normalform(T) & length(C) = 1 →
        ( ∀ i)(C = add(i) →
        /*L*/  L(T.C)
        ) &
        (C = remove →
        /*L*/  (L(T.C) ↔ T ≠ e) &
        /*E*/  (L(T.C) → ( ∀ T1,i) (T = add(i).T1 → T.C ≡ T1))
        ) &
        (C = front →
        /*L*/  (L(T.C) ↔ T ≠ e) &
               (L(T.C) →
        /*E*/      T.C ≡ T &
        /*V*/      ( ∀ T1,i) (T = add(i).T1 → V(T.C) = i))
        )
    )
```

Fig. 5. New queue specification.

obvious association between *push*, *pop*, and *top*, and *add*, *remove*, and *front*, respectively.

The new specifications differ in only two lines:

- In the equivalence section, *pop* differs from *remove*: *pop* eliminates the newest (rightmost) *push* while *remove* eliminates the oldest (leftmost) *add*.
- In the value section, *top* differs from *front: top* returns the parameter from the newest (rightmost) *push* while *front* returns the parameter from the oldest (leftmost) *add*.

In contrast, the original stack and queue specifications (see Fig. 1 and 2) are quite dissimilar, and mask the single essential difference between stack and queue behavior.

C. A Traversing Stack Specification

In this section we present a more complex example, a *traversing stack* (*tstack*), to illustrate the power of the methodology. As in the *stack* module, *tstack* permits new elements to be added or deleted only at the top of the stack. However, *tstack* provides read access to elements below the top of the stack. The module syntax is shown in Fig. 6. Informally, the semantics are as follows. If no calls to *down* or *to_top* are made, then *push*, *pop*, and *current* operate exactly as do the standard *push*, *pop*, and *top*, respectively. *Down* causes the next lower element to become the one returned by *current*, and *to_top* causes the element on the top of the stack to be returned by *current*. *Push* and *pop* are illegal unless the current element is the top element. *Pop*, *down*, and *current* are illegal if the stack is empty. *Down* is also illegal if there is no element below the current one.

We choose the normal form to be any number N of *push* calls followed by fewer than N *down* calls. We have thus

```
SYNTAX
    push:    integer;
    pop:     ;
    down:    ;
    current: → integer;
    to_top:  ;
```

Fig. 6. Traversing stack syntax.

```
PREDICATES
  normalform(T) ↔ (∃ P,D) (T = e |
     (T = P.D & all(push,P) & all(down,D) & length(P) > length(D)))

  parse(T1,C1,T2,D) ↔ (T1.C1.T2.D = e |
     (all(push,T1.C1.T2) & all(down,D) &
     length(C1) = 1 & length(T2) = length(D)))

  all(C,T) ↔ count(C,T) = length(T)
```

Fig. 7. Traversing stack predicates.

combined the *stack* normal form, representing the stack contents, with a sequence of *down* calls, representing the position of the current element. For example, for the stack contents

4
3
2 ← current element
1

the corresponding normal form trace is

push(1).*push*(2).*push*(3).*push*(4).*down*.*down*

The *normalform* predicate is shown in Fig. 7 and makes use of the simple predicate *all*.

In the *tstack* semantics, we use the *parse* predicate to split a normal form trace into the following four parts:

- T1: the *push* calls associated with those elements below the current element.
- C1: the *push* call associated with the current element.
- T2: the *push* calls associated with those elements above the current element.
- D: the *down* calls.

For example, for the normal form trace just considered

T1 = *push*(1), *C1* = *push*(2),
T2 = *push*(3).*push*(4), D = *down*.*down*

The parse predicate (see Fig. 7) expresses this structure. By providing easy access to the "current *push*" and the sequence of *down* calls in a normal form trace, the *parse* predicate makes the semantics shorter and easier to write and read. The result is a semantics section (see Fig. 8) that is quite similar to that of *stack*.

D. Specification Experience

In addition to the modules presented in this paper, the trace specification methodology has been used to respecify all five modules (*stack* and *queue* are the first two) originally specified by Parnas and Bartussek [2], and various other modules, including a relatively complex graph

```
SEMANTICS
  (∀ T,C) (normalform(T) & length(C) = 1 →
  (∀ T1,C1,T2,D) (T = T1.C1.T2.D & parse(T1,C1,T2,D) →
    (∀ i) (C = push(i) →
    /*L*/  L(T.C) ↔ D = e
    ) &
    (C = pop →
    /*L*/  (L(T.C) ↔ D = e & C1 ≠ e) &
    /*E*/  (L(T.C) → T.C ≡ T1)
    ) &
    (C = down →
    /*L*/  L(T.C) ↔ normal_form(T.C)
    ) &
    (C = current →
    /*L*/  (L(T.C) ↔ C1 ≠ e) &
           L(T.C) → (
    /*E*/     T.C ≡ T &
    /*V*/     (∀ i) (C1 = push(i) → V(T.C) = i))
    ) &
    (C = to_top →
    /*L*/  L(T.C) &
    /*E*/  (L(T.C) → T.C ≡ T1.C1.T2)
    )
  ))
```

Fig. 8. Traversing stack semantics.

traversal module. Also, significant experience has been gained writing communications protocol specifications. This work is described in detail elsewhere [13], [14].

IV. Trace Models

In this section we describe McLean's work on models of trace specifications and present a method for writing models in Lisp from specifications written using the methodology described in Section III.

A. McLean Models

A *model* for a trace specification consists of a tuple *D* of *domains* and an *interpretation function I* that assigns to each meaningful element of the trace language an element of a domain of *D* [18]. The domains in the tuple *D* are sets of values, such as the integers or the set of all strings on some alphabet. Meaningful elements of the trace language include the predicate *L* and the function *V*. We say that a specification is *consistent* if it is impossible to derive a contradiction from its assertions. McLean provides a formal definition of model, shows that a trace specification is consistent if and only if it has a model, and proposes model-building as the preferred method for proving the consistency of a trace specification.

McLean presents models for two modules, variants of the *stack* and *queue* modules discussed above. For the purposes of this paper, two aspects of the model are of interest: 1) the domain, in the tuple *D*, corresponding to traces, and 2) $I(L)$ and $I(V)$, the interpretations assigned to the predicates *L* and *V*, respectively. In McLean's *stack* model, the trace domain consists of the set of all character strings corresponding in the obvious way to sequences of calls. For example, the string "*push*(6).*top*.*pop*", which is a trace domain element, corresponds to the trace *push*(6).*top*.*pop*. Definitions for $I(L)$ and $I(V)$ are based on a simple algorithm, expressed in pseudocode. The al-

gorithm, particular to this model, takes a trace T and removes *top* calls and matching *push* and *pop* calls, transforming T into an equivalent trace T_{NF} containing only *push* calls. $(I(L))(T)$ is defined to be true if and only if T can be so transformed, and $(I(V))(``T.top")$ is defined to be the value pushed by the last call in T_{NF}.

For trace specifications written in an ad hoc fashion, models must also be constructed ad hoc. Considerable insight into each specification is required to produce normalization algorithms of the type described above. However, for specifications written using the trace methodology, models can be constructed manually in a straightforward fashion, converting the specification one line at a time. Insight into the meaning of each specification is therefore unnecessary (and perhaps undesirable!) for the model builder. Below, we motivate our choice of modeling language, describe the conversion from specification to model and present three examples.

B. Lisp Models

We have chosen to write in Lisp [23], for the following reasons:

- Since Lisp is a formal language, a Lisp model is formal, avoiding the vagueness of prose or pseudocode.
- Since Lisp is a programming language, a Lisp model is directly executable (though perhaps inefficiently). All the Lisp code presented in this paper has been run and tested.
- Traces are naturally represented as lists and lists are easily manipulated in Lisp.
- Lisp-based automatic theorem proving is available to assist in proving assertions about Lisp models [3].

For those unfamiliar with Lisp, the text should still be comprehensible, but the code in the Appendix may not. Note that a `fixed width font` is used for all Lisp objects.

Lisp models can constructed in a straightforward manner from specifications written using the trace specification methodology. For the trace domain we use Lisp lists. In the model, a trace is a (possibly empty) list of calls and a call is a (nonempty) list consisting of a procedure or function name followed by actual parameters of the appropriate number and type. For example, the trace *push*(1).*push*(2).*pop*.*top* becomes, in the model `((push 1) (push 2) (pop) (top))`. We define $I(L)$ and $I(V)$ using the Lisp functions `L` and `V`, respectively. `L` takes a trace parameter T and returns true or false, according to whether T is legal or not. `V` takes a trace parameter T, assumed to be legal and ending in a function call, and returns $V(T)$. Having expressed `L` and `V` in terms of trace models, we describe how to implement them to represent a model for a given trace specification.

Our trace specifications are expressed in terms of the legality, equivalence, and value of a single "new" call C, following a normal form "history" T. We imitate this structure in our Lisp models by writing functions `lp`, `nf`, and `v`, each taking T and C as parameters. `lp` returns true or false according to whether $T.C$ is legal, `nf` returns a normal form trace equivalent to $T.C$, and `v` returns $V(T.C)$. One problem remains: `lp`, `nf`, and `v` operate only on normal form traces, while `L` and `V` must also handle traces not in normal form. Our solution is the function `nfdriver`, that recursively applies `lp` and `nf` to detect an illegal trace and transform a legal one into its normal form equivalent. The functions `L`, `V` and `nfdriver`, together with several utility functions, are shown in the Appendix under "Model Independent Functions." These functions are the same for all models we have written. The `lp`, `nf`, and `v` functions are different for each model; versions for *stack* and *tstack* are presented below.

Throughout the remainder of Section IV, we follow the convention that `tr`, `nftr`, and `c` are Lisp variables representing traces, normal form traces, and traces of length one, respectively. Note that, in Lisp, `'t` represents logical true and `nil` logical false. When writing the model specific functions, we choose Lisp constructions that follow the specification as closely as possible, even when significant time or space inefficiencies result and could be avoided by "clever programming." Thus, we emphasize confidence in model correctness rather than model efficiency. When appropriate, the Lisp model could be improved, or translated to a more efficient language.

C. A Stack Model

The `lp`, `nf`, and `v` functions (Appendix—"Stack Functions") are derived directly from the *stack* specification of Fig. 4. `lp` is generated from the lines dealing with legality (labeled /*L*/); `nf` is generated from the lines dealing with equivalence (labeled /*E*/); and `v` is generated from the lines dealing with value (labeled /*V*/). We illustrate the operation of the *stack* model by listing the `nfdriver` calls and actual parameters that result from a call to L. If we execute

```
(L '((push 1) (push 2) (pop) (top)))
```

then the `nfdriver` calls will be

```
(nfdriver nil ((push 1) (push 2) (pop) (top)))
(nfdriver ((push 1)) ((push 2) (pop) (top)))
(nfdriver ((push 1) (push 2)) ((pop) (top)))
(nfdriver ((push 1)) ((top)))
(nfdriver ((push 1)) nil)
```

and the `nfdriver` call invoked directly by `L` will return `'t`, indicating that *push*(1).*push*(2).*pop*.*top* is a legal trace. If, however, we execute

```
(L '((push 1) (pop) (top)))
```

then the `nfdriver` calls will be

```
(nfdriver nil ((push 1) (pop) (top)))
(nfdriver ((push 1)) ((pop) (top)))
```

The last `nfdriver` call shown will call

```
(lp nil ((top)))
```

and `lp` will return `nil`, causing `nfdriver` to return `'error`, indicating that *push*(1).*pop*.*top* is an illegal trace. Calls to `V` are handled in a similar fashion, except that only the

head of the trace is transformed to normal form and then this normal form trace and the tail of the original trace are passed to v. For example, (V '((push 1)(push 2)(pop)(top))) will return 1.

D. A Traversing Stack Model

The *stack* and *tstack* specifications are quite similar except for the presence of the *parse* predicate in the latter. Appropriately, the two models are similar except for the parse function in the *tstack* model. In the specification, the *parse* predicate is used to assert that if a trace T is divided into the subtraces $T1$, $C1$, $T2$, and D, then certain assertions hold on the subtraces. In the model, the parse function takes the parameter nftr and finds the four subtraces, so that new traces can be constructed from them. The fact that specifications assert constraints on traces and models search for traces to fit those constraints clearly characterizes the difference between our specifications and models.

The parse function (Appendix—"Traversing Stack Parse Function") is implemented using two other functions. Subtrace takes the parameters, tr, s, and l and returns the subtrace of tr starting at position s and l calls in length. The count Lisp function operates identically to the *count* function built into the specification language. With these two functions, parse is quite simple to implement: by using count to determine the number of down calls, the four subtraces are easily extracted with subtrace. With parse available, the lp, nf, and v functions (Appendix—"Traversing Stack Functions") are coded directly from the specification. Essentially, the tstack model is just the stack model with calls to the parse function added.

E. Specifications, Models, and Totalness

One of the strengths of the trace approach is that models and specifications are distinct, while model-based specification methods blur or eliminate this distinction. This section introduces the term *total* and presents a nontotal specification and one of its models.

A specification is total if, for each legal trace ending in a function call, exactly one value for the trace can be derived from the specification [18]. As noted before, a specification is consistent if it has a model. If it is both consistent and total, it has at least one model and all of its models behave identically (with respect to function call return values). If it is consistent and nontotal, it has models with different behavior. The three specifications presented above are total. We now present an example of a nontotal specification.

The *unique* module provides unique integers for such applications as automatic file or variable naming. *Unique* supports the single call *getint*, which takes no parameters and returns an integer value. Any value is acceptable, as long as it has not been returned by a previous *getint* call. In the specification (see Fig. 9), we choose the normal form to be all traces: no other choice is possible. The semantics section is straightforward. *T.getint* is always legal and *getint* always returns a "new" value. No equivalence assertion is needed because *T.getint* is always normal form itself.

```
NAME
    unique

SYNTAX
    getint:  → integer;

PREDICATES
    normalform(T) ↔ true

SEMANTICS
    ( ∀ T,C) (normalform(T) & length(C) = 1 →
        (C = getint →
        /*L*/  L(T.C) &
        /*V*/  (L(T.C) → ( ∀ T1) (prefix(T1,T) → V(T1) ≠ V(T.C)))
        )
    )
```

Fig. 9. Unique specification.

In our *unique* model (Appendix—"Unique Model"), the first *getint* call returns one and each successive call returns one greater than the previous call, via the length function call in v. Infinitely many other choices are available. For example we may redefine v as

(defun v (nftr c) (plus (length nftr) *K*))

where K is any integer constant other than 0. The result is a model of the specification that behaves differently from that of Fig. 13, returning K, $K + 1$, $\cdots$ instead of 0, 1, $\cdots$.

F. Other Models

Using the approach described above, we have written Lisp models (and hence, consistency proofs by construction) for all the examples in the original traces paper [2]. Of the last example in that paper, Parnas and Bartussek say "demonstration of consistency is more complex . . . such a proof is beyond the scope of this paper . . . " [2].

V. A Comparison: Trace and Algebraic Specifications

The algebraic specification method is a well-known technique for formal software specification [8], [9], [11]. A comparison between the trace and algebraic methods provides insight into the strengths and weaknesses of each.

We begin by noting that the two methods are similar in three important ways. First the goal of both methods is the same: the implementation-independent specification of a software module in terms of calls and return values. Second, both the trace and algebraic methods have solid formal foundations, the former based on first order logic and the latter on heterogeneous algebra. And third, both methods are in fact "algebraic," because first order logic is a cylindric algebra [18].

The most important difference between the two methods is that algebraic specifications are based on the *type of interest*, while trace specifications are based on call sequences. For a module where the type of interest is not observable to its users (which is the case for all the mod-

ules specified in this paper) an algebraic specification must be written in terms of objects not visible to the module's users. We see this as a fundamental weakness in the algebraic approach. Consider an algebraic specification of the *stack* module (Appendix—"Algebraic Stack Specification"). The *push* call *must* take a stack parameter and return a stack value. The specification *must* contain a *newstack* call returning an "empty stack." And, the specification provides no representation for the sequence of events represented by the trace *push*(1).*top*.*top* [18]. The obvious choice of *top*(*top*(*push*(*newstack*,1))) is not available because the inner *top* returns 1 while the outer *top* requires a parameter of type stack. The trace method does not have these problems: the type of interest can be made visible or not, according to the specifier's intentions.

We see three other weaknesses of the algebraic method, relative to traces. First, many algebraic specifications require "hidden functions": calls (in addition to calls such as *newstack*) that are not visible to the module user but are essential to express the semantics of the visible operations. For example, the *tstack* module cannot be specified without hidden functions. In fact, early *tstack* specification attempts generated heated debate as to whether an *infinite* number of algebraic axioms were required [15]-[17]! We have not used and see no reason to use hidden functions in trace specifications. Second, we see no way to extend the algebraic method to specify interprocess communication. While published research describes algebraic specification of communications protocols [22], this work makes heavy use of explicit state variables and so is not algebraic in the standard sense. However, traces extend naturally to handle interprocess communication [13, 14]. Finally, algebraic specifications do not permit existential quantification: all variables are assumed to be universally quantified. As a result, there appears to be no algebraic specification for modules such as *unique* [18].

On the other hand, while the application of the trace method is relatively limited thus far, there has been considerable experience with algebraic specification [4], [7], [10]. Also, algebraic specifications appear to be more suitable for machine processing, and even compilation, than trace specifications. Among the support software already implemented is the AFFIRM theorem prover [12]. Finally, languages such as HOPE [5] provide programming languages very close in appearance to algebraic specifications. A the present time, support software for the trace methodology has not been developed.

VI. Conclusions

The specification methodology described in this paper is important because it makes the trace language significantly more useful. The methodology makes trace specifications easier to read, write, and change, and provides insight into the similarities and differences between modules. Recent specification of complex communications protocols depended heavily on the methodology and increased our confidence in its value. Finally, the methodology supports the straightforward building of models which serve as consistency proofs and executable prototypes.

Concerning model building, there are several areas where more work is needed. Currently the lp, nf, and v functions are coded manually. By referring to the trace specification, this is a straightforward task. We would prefer, however, to have the model constructed partially or entirely automatically. In Section IV-B, we mentioned that Lisp-based automatic theorem proving tools could assist in proving assertions about Lisp models. This claim should be explored further. Based on manual proofs we have already completed, we would like to be able to prove mechanically that all the assertions of a specification are true for its (proposed) model. Preliminary work has been done on manual and automatic conversion of trace specifications into Prolog [6], [19], but more work is needed.

Appendix

Model Independent Functions

```
(defun L (tr)
   (cond ((eq (nfdriver nil tr) 'error) nil)
      ('t 't)))

(defun V (tr) (v (nfdriver () (head tr)) (tail tr)))

(defun nfdriver (nftr tr)
   (cond ((null tr) nftr)
      ('t(cond ((lp nftr (list (car tr))) (nfdriver
               (nf nftr (list (car tr))) (cdr tr)))
            ('t 'error)))))
(defun callnamep (tr n)
   (cond ((and (eq (length tr) 1) (eq (caar tr) n) 't))
      ('t nil)))
(defun head (tr)
        (cond   ((null (cdr tr)) nil)
      ('t (cons (car tr) (head (cdr l))))))
(defun tail(tr)
   (cond ((eq tr nil) nil)
      ((eq (cdr tr) nil) (list (car tr)))
      ('t (tail (cdr tr)))))
(defun emptytr (tr) (eq tr nil))
```

Stack Functions

```
(defun lp (nftr c)
   (cond ((callnamep c 'push) 't)
      ((and (callnamep c 'pop) (not (emptytr nftr)) 't))
      ((and (callnamep c 'top) (not (emptytr nftr)) 't))
      ('t nil)))

(defun nf (nftr c)
   (cond ((callnamep c 'push) (append nftr c))
      ((callnamep c 'pop) (head nftr))
      ((callnamep c 'top) nftr)))

(defun v (nftr c) (cadar (tail nftr)))
```

Queue Functions

```
(defun lp (nftr c)
   (cond ((callnamep c 'add) 't)
      ((and (callnamep c 'remove)
         (not (emptytr nftr)) 't))
      ((and (callnamep c 'front)
         (not (emptytr nftr)) 't))
      ('t nil)))
```

```
(defun nf (nftr c)
   (cond ((callnamep c 'add) (append nftr c))
       ((callnamep c 'remove) (cdr nftr))
       ((callnamep c 'front) nftr)))

(defun v (nftr c) (cadar nftr))
```

Traversing Stack Parse Function

```
(defun parse (tr)
   (list (pt1 tr) (pc1 tr) (pt2 tr) (pd tr)))
(defun pt1 (tr) (subtrace tr
   1 (- (length tr) (* 2 (count 'down tr)) 1)))
(defun pc1 (tr) (subtrace tr
   (- (length tr) (* 2 (count 'down tr))) 1))
(defun pt2 (tr) (subtrace tr
   (+ (- (length tr) (* 2 (count 'down tr))) 1)
   (count 'down tr)))
(defun pd (tr) (subtrace tr
   (+ (- (length tr) (count 'down tr)) 1)
   (count 'down tr)))
(defun subtrace (tr s l)
   (cond ((or (lessp s 1) (lessp l 1)) nil)
      ((eq s 1)
         (append (list (car tr))
            (subtrace (cdr tr) 1 (- l 1))))
      ('t (subtrace (cdr tr) (- s 1) l))))
(defun count (callname tr)
   (cond ((null tr) 0)
      ('t(cond
            ((eq (caar tr) callname)
               (+ 1 (count callname (cdr tr))))
            ('t (count callname (cdr tr)))))))
```

Traversing Stack Functions

```
(defun lp (nftr c) (lp1 (parse nftr) c))
(defun lp1 (pnftr c)
   (cond ((callnamep c 'push) (emptytr (d pnftr)))
      ((callnamep c 'pop)
         (and (not (emptytr (c1 pnftr)))
            (emptytr (d pnftr))))
      ((callnamep c 'down) (not (emptytr (t1 pnftr))))
      ((callnamep c 'current)
         (not (emptytr (c1 pnftr))))
      ((callnamep c 'to_top) 't)
      ('t nil)))
(defun nf (nftr c) (nf1 nftr (parse nftr) c))
(defun nf1 (nftr pnftr c)
   (cond ((callnamep c 'push) (append nftr c))
      ((callnamep c 'pop) (t1 pnftr))
      ((callnamep c 'down) (append nftr c))
      ((callnamep c 'current) nftr )
      ((callnamep c 'to_top)
         (append (t1 pnftr) (c1 pnftr) (t2 pnftr)))
      ('t nil)))
(defun v (nftr c) (cadar (c1 (parse nftr))))
(defun t1 (pnftr) (car pnftr))
(defun c1 (pnftr) (cadr pnftr))
(defun t2 (pnftr) (caddr pnftr))
(defun d (pnftr) (cadddr pnftr))
```

Unique Functions

```
(defun lp (nftr c)
   (cond ((callnamep c 'unique) 't)))

(defun nf (nftr c)
   (cond ((callnamep c 'unique) (append nftr c))))

(defun v (nftr c) (length nftr))
```

Algebraic Stack Specification

syntax
- push: stack × integer → stack
- pop: stack → stack
- top: stack → integer
- newstack: → stack

semantics
- pop(push(s,i)) = s
- top(push(s,i)) = i

restrictions
- s = newstack → failure(pop,s)
- s = newstack → failure(top,s)

Acknowledgment

We would like to thank R. N. Horspool, Y. Wang, and G. Yang for help in developing the Lisp models.

References

[1] A. L. Ambler *et al.*, "GYPSY: A language for specification and implementation of verifiable programs," in *Proc. ACM Conf. Language Design for Reliable Systems*, ACM, Mar. 1977, pp. 1–10.

[2] W. Bartussek and D. L. Parnas, "Using assertions about traces to write abstract specifications for software modules," in *Proc. Second Conf. European Cooperation in Informatics*. New York: Springer-Verlag, 1978.

[3] R. S. Boyer and J. S. Moore, "Proving theorems about LISP functions," *J. ACM*, vol. 22, no. 1, pp. 129–144, Jan. 1975.

[4] R. M. Burstall and J. A. Goguen, "An informal introduction to specifications using CLEAR," in *The Correctness Problem in Computer Science*, vol. 13. New York: Academic, 1981, pp. 185–213.

[5] R.M. Burstall, D. B. Macqueen, D. T. Sannella, "HOPE: An experimental applicative language," in *Conf. Rec. 1980 LISP Conf.*, Aug. 1980, pp. 136–143.

[6] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. New York: Springer-Verlag, 1981.

[7] N. H. Gehani, "Specifications: Formal and informal—A case study," *Software—Practice and Experience*, vol. 12, pp. 185–213, Jan. 1982.

[8] J. A. Goguen and J. J. Tardo, "An introduction to OBJ: A language for writing and testing formal algebraic program specifications," in *Proc. 1976 Int. Conf. Parallel Processing*, IEEE, 1976, pp. 176–189.

[9] J. Guttag, "Notes on type abstraction," *IEEE Trans. Software Eng.*, vol. SE-6, no. 1, pp. 13–23, Jan. 1980.

[10] J. V. Guttag and J. J. Horning, "Formal specification as a design tool," in *Conf. Rec. Seventh Annu. ACM Symp. Principles of Programming Languages*, ACM, 1980.

[11] —, "The algebraic specification of abstract data types," *Acta Inform.*, vol. 10, no. 1, pp. 27–52, 1978.

[12] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *Commun. ACM*, vol. 21, no. 12, pp. 1048–1064, Dec. 1978.

[13] D. M. Hoffman, "Trace specification of communications protocols," Ph.D. dissertation, Dep. Comput. Sci., Univ. North Carolina, 1984.

[14] —, "The trace specification of communications protocols," *IEEE Trans. Comput.*, vol. C-34, no. 12, Dec. 1985.

[15] M. E. Majster, "Limits of the 'algebraic' specification of abstract data types," *SIGPLAN Notices*, vol. 12, no. 10, pp. 37–42, Oct. 1977.

[16] —, "Comment on a note by J. J. Martin in SIGPLAN Notices, vol. 12, no. 10," *SIGPLAN Notices*, vol. 13, no. 1, pp. 22–23, Jan. 1978.

[17] J. J. Martin, "Critique of Mila E. Majster's 'Limits of the algebraic specification of abstract data types,'" *SIGPLAN Notices*, vol. 12, no. 12, pp. 28–29, Dec. 1977.

[18] J. McLean, "A formal method for the abstract specification of software," *J. ACM*, vol. 31, no. 3, July 1984.

[19] J. M. McLean, D. Weiss, and C. Landwehr, "Executing trace specifications using Prolog," Naval Research Lab., Tech. Rep. 1985.

[20] D. L. Parnas, "The use of precise specifications in the development of software," in *Proc. IFIP Congress 1977*. Amsterdam, The Netherlands: North-Holland, 1977.

[21] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Trans. Software Eng.*, vol. SE-12, no. 2, pp. 251–257, Feb. 1986.

[22] C. A. Sunshine *et al.*, "Specification and verification of communication protocols in AFFIRM," *IEEE Trans. Software Eng.*, vol. SE-8, no. 5, pp. 460–489, Sept. 1982.

[23] R. Wilensky, *LISPcraft*. Norton, 1984.

Daniel Hoffman (S'83–M'84) received the B.A. degree in mathematics from the State University of New York, Binghamton, in 1974, and the M.S. and Ph.D. degrees in computer science from the University of North Carolina, Chapel Hill, in 1981 and 1984, respectively.

From 1974 to 1979 he worked as a commercial Programmer/Analyst. He is currently an Assistant Professor of Computer Science at the University of Victoria, B.C., Canada. His research area is software engineering, with emphasis on software specification.

Richard Snodgrass received the B.A. degree in physics from Carleton College, Northfield, MN, in 1977 and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA in 1982.

He is currently an Assistant Professor at the Department of Computer Science, University of North Carolina, Chapel Hill. His research interests include temporal databases, programming environments, and distributed systems. He is the director of the SoftLab project, which is concerned with the implementation of programming environments, database management systems, and operating systems.