

PROTOTYPES OF TRACE SPECIFICATIONS

by

YABO WANG

B.Sc., University of Science & Technology of China, 1982

ACCEPTED

ACULTY OF GRADUATE STUDIES

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in the Department of Computer Science

DEAN

We accept this thesis as conforming
to the required standard

Dr. Daniel M. Hoffman

Dr. William W. Wadge

Dr. Charles G. Morgan

Dr. Gary G. Miller

© YABO WANG, 1988

University of Victoria
April 1988

*All rights reserved. This thesis may not be reproduced
in whole or in part, by mimeograph or other means,
without the permission of the author.*

QA 76.76

V47 W35

Supervisor: Professor Daniel M. Hoffman

ABSTRACT

In large and complex software systems, precise abstract software specification can play a very important role. For specifications written in a formal language, the existence of a prototype provides significant benefits. Although such prototypes are usually inefficient, they can help the users and specifier gain insight. To be practical, it is crucial that these prototypes can be constructed cheaply and with confidence in their correctness. This thesis presents a scheme for quickly and reliably constructing Prolog prototypes from trace specifications. The interfaces that make prototypes easy to access are also described.

A prototype thus built can help the users and specifier to understand and verify the behavior described by a specification. However, not until the correctness of the prototype can be proved, can we be sure that its behavior meets its specification. For prototypes built in an ad hoc manner, the proofs must be ad hoc as well. For prototypes built by our scheme, we present a corresponding proof scheme for proving prototype correctness.

Some software modules require some of their return values to be non-deterministic. It is desirable that we can verify that their specifications reflect the correct module behavior. A straightforward scheme to check whether a specification is deterministic is discussed.

Examiners:



Professor Daniel M. Hoffman



Professor William W. Wadge



Professor Charles G. Morgan



Professor Gary G. Miller

Table of Contents

Abstract	ii
Table of Contents	iv
List of Figures	vii
Acknowledgements	ix
1. Introduction	1
2. Background	4
2.1 Operational Specification	5
2.2 SCR Method	5
2.3 Algebraic Specification	7
2.4 Trace Specification	10
2.5 Comparisons	12
3. Trace Methodology	14
3.1 Trace Methodology	14
3.2 Specification Examples	16
3.2.1 Stack	16
3.2.2 Unique	17
3.2.3 Priority Queue	17
3.2.4 Traversing Stack	18
	iv

3.2.5 Binary Tree	21
4. Prototype Building	27
4.1 Scheme	27
4.2 Prototype Examples	29
4.2.1 A Stack Prototype	29
4.2.2 A Traversing Stack Prototype	30
4.2.3 A Binary Tree Prototype	32
4.3 Prototype User Interfaces	33
4.3.1 The Prolog Interface	34
4.3.2 The C Interface	36
5. Prototype Correctness Proof	38
5.1 McLean's Model	39
5.1.1 Model Definition	39
5.1.2 Stack Model	39
5.1.3 The Stack Model's Correctness	41
5.2 Our Model	42
5.2.1 Proof Scheme	43
5.2.2 Prolog Prototype and Trace Model	43
5.2.2 The Correctness of a Model	47
5.3 Stack Prototype Correctness Proof	49
6. Deterministic And Non-Deterministic Specifications	56

6.1 Deterministic Specification	56
6.2 Non-Deterministic Specifications	57
7. Conclusions and Future Work	60
Bibliography	62
Appendix A	64
Appendix B	76

List of Figures

Figure 1. Bstack Operational (Pascal) Specification	5
Figure 2. Bstack SCR Specification	7
Figure 3. Bstack Algebraic Specification	9
Figure 4. Stack Trace Specification	11
Figure 5. New Stack Trace Specification	16
Figure 6. Unique Trace Specification	17
Figure 7. Priqueue Trace Specification	18
Figure 8. Traversing Stack Syntax	18
Figure 9. Traversing Stack Predicates	20
Figure 10. Traversing Stack Semantics	21
Figure 11. Tree Syntax	23
Figure 12. Predicates Specifying A Tree	24
Figure 13. Tree Normal Form Predicate	24
Figure 14. A Tree Example	25
Figure 15. Del Semantics	25
Figure 16. Module Independent Procedures	28
Figure 17. Stack Prototype	29
Figure 18. Parse Procedures	31
Figure 19. Traversing Stack Prototype	32

Figure 20. Tree Prolog Procedures	33
Figure 21. Del Prolog Procedure	33
Figure 22. Stack Prolog Interface	35
Figure 23. Requirements for Function I	39
Figure 24. A Stack Trace Specification	40
Figure 25. Common Procedures in PQ1 and PQ2	57
Figure 26. Parse Procedures in PQ1	58
Figure 27. Parse Procedures in PQ2	58

Acknowledgements

I wish to thank Professor Daniel M. Hoffman for suggesting the topic of this thesis and for his valuable guidance of my research.

CHAPTER 1

INTRODUCTION

A key problem in the development of large software systems is reducing the amount of complexity or detail that must be considered at any one time. 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*, a precise description of the essential behavior of the modules [1], 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.

There are important benefits which stem from writing specifications. A common problem in system construction is that the behavior expected by a user of a module may not be the same as what the module provides. This problem can be avoided, or at least greatly reduced, if the design phase provides specifications of the module. The module specification serves to document the intended module behavior and to communicate this behavior to both the implementor of the module and to programmers who will use that module in implementing other modules. Without a precise specification, it would be meaningless to say that a piece of software meets its intended requirements. Further, if module specifications are written in a formal language, then a thorough mathematical study of the specifications is possible. This enables us to dis-

cover certain forms of inconsistency or incompleteness in specifications before implementation, reducing the cost of the software development greatly. The range of benefits are much wider than this; they relate to methods of programming, to possible approaches to validation and verification of programs, to the management and control of large software projects, and to the maintenance of large pieces of software.

There are a variety of specification methods. Some common approaches reveal a rough description of the internal structure of a module. Others reveal a description of a “hypothetical” implementation of the module. In the first case, the system becomes hard to change since the correctness of the interfacing programs may depend directly upon the internal information that was revealed. In the second case, the program may never work correctly, because the correctness depends on some feature of the “hypothetical” implementation that was never provided in the real one [2]. In contrast to those approaches, it is proposed that module specifications should be written without any mention of internals: all statements are in terms of the access programs of the module and could be verified by a module user.

Based on this “black-box” idea, Parnas and Bartussek invented the *trace* specification language [3]. McLean has established a formal basis for the trace language [4]. Hoffman has developed a methodology for handling complex trace specifications [1]. The background of the trace specification method and a comparison with other specification methods are presented in Chapter 2.

Hoffman’s trace methodology was developed to deal with large and complex modules. To gain some experience with using the methodology, we have specified some fairly large and complex modules, such as the *tree* and the *shortest-path*. The results are quite satisfactory. In Chapter 3, we present Hoffman’s new trace methodology and introduce a set of trace

specifications used as examples in the later chapters.

To gain insight into a specification before it is actually implemented, models are often used to prove that the specification is consistent and to provide prototypes for the user and the designer [8]. For specifications written in an ad hoc manner, prototype building must also be ad hoc. However, for specifications written in the trace methodology, we have developed a method to uniformly build prototypes in Prolog [6]. Our prototype building method is presented in Chapter 4.

Since the prototype building method in Chapter 4 is uniform across modules, the prototype correctness proof is also simple and uniform. The prototype correctness proof scheme is presented in Chapter 5.

For some modules, we want the return value of every program to be unique, i.e., deterministic, in the specification. For some other modules, however, the implementor is purposely given some freedom in the choice of its values (see the *priority queue* and *spath* specifications in the Appendix). Thus, it is desirable to be able to show that a module specification meets our intuitive understanding of the module's function. In Chapter 6, we discuss a scheme for proving that a trace specification is deterministic or non-deterministic. And in Chapter 7, we present our conclusions, contributions, and further research areas.

CHAPTER 2

BACKGROUND

Software specifications can have various degrees of formality. At the informal end of the spectrum, the specifications can be expressed in some combination of natural language, diagrams and standard mathematical notations. Although such specifications may be intuitive and easily understood, they are often incomplete, vague and ambiguous. Specifications written in prose belongs to this class. At the other end are specifications written entirely in a language with an explicitly and precisely defined syntax and semantics, such as the operational [5,7], SCR [9,13], algebraic [10,11,12], and trace [1,3,4] methods. For these *formal* specifications, the well-defined and unambiguous semantics of the specification languages eliminate or lessen the possibility of misinterpretation.

We say a specification is *abstract* if it is written solely in terms of the observable behavior of the module, i.e., if it is independent of any implementation, especially any data structure in the module. The SCR, algebraic, and trace methods are all abstract in this sense.

In Section 2.1 - 2.4, we survey the following four specification methods: (1) operational specification method, (2) the SCR method, (3) the algebraic method, and (4) the trace method. The methods are compared in Section 2.5. We illustrate the use of these methods by specifying a bounded integer stack (hereafter *bstack*) module. We assume that the size of the *bstack* is 5. *bstack* supports three calls:

push takes an integer parameter and pushes it onto the stack.

pop takes no parameter and returns no value, but removes the the top stack element.

top returns the value of the top stack element.

pop and *top* are illegal on an empty stack and *push* on a full stack is illegal.

2.1. Operational Specification

In an operational specification, the relationship between the input and the output of a module is described by explicitly giving a program *P*. Then, any program which has the same input and output behavior as *P* is considered as a correct implementation of *P*. An operational specification written in Pascal for the *bstack* is shown in Figure 1.

In this specification, *stack* is declared as an integer array of the stack size and *ptr* is the index of the top element in the stack.

2.2. SCR Method

The SCR (Software Cost Reduction) method [13] uses a standard organization for specifying abstract module interfaces. Each module provides a number of functions (*set calls*) which can cause changes in the module's state, and other functions (*get calls*) which give a user program information about that state.

The major sections used in an SCR specification are:

Syntax

A table of programs on the module's interface. For each access program, the table provides its name, and, for each of its parameters, states its type and indicates whether it is an input or output parameter. Finally, the table lists the names of the

Figure 1. Bstack Operational (Pascal) Specification

```

const
    STACKSIZE = 5;
var
    stack: array [1..STACKSIZE] of integer;
    ptr: integer;
procedure push(i:integer);
begin
    if (ptr <> STACKSIZE) then
        begin
            ptr := ptr + 1;
            stack[ptr] := i
        end
    else
        writeln('ERROR: stack is full')
    end;
procedure pop;
begin
    if (ptr <> 0) then
        ptr := ptr - 1
    else
        writeln('ERROR: stack is empty')
    end;
function top:integer;
begin
    if (ptr <> 0) then
        top := stack[ptr]
    else
        writeln('ERROR: stack is empty')
    end;
end;

```

exceptions that calling the program may, under some circumstances, generate.

Effects

For each set or set-get call S , there is a subsection under the Effects. The predicate under each subsection describes the effect that calling S has on the output parameter values of other get calls.

Exceptions

For each exceptions listed in the Syntax, specify conditions under which the exception occurs.

In Figure 2, we give an SCR specification for the *bstack*. Since *top* is the only get call, all the effects that calling set calls (*push* and *pop*) have on the module must be described by *top* alone. However, this is not possible because each *push* call “hides” the previous *top* value. The only solution to this problem is to introduce hidden programs into the specification: here we use *depth* and *ith*. A hidden program is a get call artificially added to a specification to record effects not immediately visible using the module’s access programs. Hidden programs are not implemented. In Figure 2, *ith* returns the value of the *i*th stack element, where elements are numbered in the order they were pushed and the bottom stack element is numbered 1. The single quote (') before or after a get program name indicates the value returned by that program before or after a certain set call is invoked, respectively. For example, in the line *depth' = 'depth + 1, ' depth* and *depth'* indicate *depth*’s value before and after *push(i)* is called, respectively.

2.3. Algebraic Specification

An algebraic specification [10] of a module consists of three parts: a syntactic specification, a semantic specification, and a restriction specification. The syntactic specification provides the names, domains, and ranges of the operations associated with the module. The semantic specification is a set of axioms which define the meaning of the operations by stating their relationships to one another. The value and operations are regarded as together forming an abstract algebra. In the restriction section, restrictions on operations are specified. A formula of the

Figure 2. Bstack SCR Specification

SYNTAX

Access Program Name	Input	Parameters Output	Exceptions
push	+integer+		%full%
pop			%empty%
top		+integer+	%empty%
Hidden Programs			
depth		+integer+	
ith	+integer+	+integer+	

EFFECTS

Initially:

depth' = 0

After push(i):

depth' = 'depth + 1 and
top' = i and
ith(depth')' = i

After pop:

depth' = 'depth - 1 and
if depth' > 0 then
top' = 'ith(depth')

EXCEPTIONS

push: (%full%, depth = 5)

pop: (%empty%, depth = 0)

top: (%empty%, depth = 0)

form $P(X) \rightarrow \text{failure}(g, X)$ states that if the operation g is invoked with arguments X such that $P(X)$ is true, then g must fail, i.e., g must not terminate normally.

An algebraic specification for the *bstack* is presented in Figure 3.

The information about the number of parameters and parameter types for *push*, *top*, and *pop* are described in the Syntax section. Note the additional functions *newstack* and *depth* in the Syntax section. *newstack* is required to create a empty stack. The * preceding *depth* in the syntactic specification indicates that *depth* is a hidden function. As for the SCR method, hidden functions are not implemented. They are artificially added when the relations among operations or the restrictions on operations can not be specified clearly by using the implemented

Figure 3. Bstack Algebraic Specification

Syntax

$\text{push: Bstack} \times \text{Integer} \rightarrow \text{Bstack}$
 $\text{pop: Bstack} \rightarrow \text{Bstack}$
 $\text{top: Bstack} \rightarrow \text{Integer}$
 $\text{newstack: } \rightarrow \text{Bstack}$
 $*\text{depth: Bstack} \rightarrow \text{Integer}$

Semantics

$\text{pop}(\text{push}(\text{stk}, \text{elm})) = \text{stk}$
 $\text{top}(\text{push}(\text{stk}, \text{elm})) = \text{elm}$
 $\text{depth}(\text{newstack}) = 0$
 $\text{depth}(\text{push}(\text{stk}, \text{elm})) = 1 + \text{depth}(\text{stk})$

Restrictions

$\text{stk} = \text{newstack} \rightarrow \text{failure}(\text{pop}, \text{stk})$
 $\text{stk} = \text{newstack} \rightarrow \text{failure}(\text{top}, \text{stk})$
 $\text{depth}(\text{stk}) = 5 \rightarrow \text{failure}(\text{push}, \text{stk}, \text{elm})$

operations alone. The reason that we introduce *depth* in Figure 3 is that the condition that a stack is full can not be specified without it.

2.4. Trace Specification

In a trace specification, the user is told only of a set of access programs on the module. Then, 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:

$$name : parameter_type \cdots parameter_type \rightarrow return_value_type$$

- (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 an illegal trace is undefined.
- (3) What is the return value of the function call at the end of a legal trace? 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 $\mid$ 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 they are defined to be *equivalent* and we write $T_1 \equiv T_2$. More precisely, any two traces T_1 and T_2 are equivalent if

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

It is also assumed that any prefix of a legal trace is legal. More formally:

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

A trace specification for the *bstack* is shown in Figure 4. In the specification, we use the functions *length* and *count*. to make the first assertion short and more readable. Informally, they are defined as:

length(T)

length(T) = n if and only if n is the number of calls in T .

count(c, T)

count(c, T) = n if and only if T has n calls with name c .

The first assertion states that any sequence of *push* calls of length less than 6 is legal trace. The second assertion states that calling *top* will not result in an error if calling *pop* does not. The next two assertions show that *push* - *pop* pairs and *top* have no observable effects on any

Figure 4. Stack Trace Specification

```

NAME
    bstack
SYNTAX
    push:  integer;
    pop:   ;
    top:   → integer
SEMANTICS
    /*1*/  (∀ T)((count(push,T)=length(T)) & length(T)≤5→L(T))

    /*2*/  (∀ T)(L(T.top)↔L(T.pop))

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

    /*4*/  (∀ T)(L(T.top)→T≡T.top)

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

```

trace. And the last assertion shows the value of *top* after any trace that ends with a *push*.

2.5. Comparisons

In all four specification methods, the syntactic specification sections are essentially the same: they all provide access program names and their input and output parameters types.

Among the four methods, the operational specifications are often easier to construct. However, the method mixes design decisions (what the module is supposed to do) and implementation decisions (how the module satisfies its requirements). Thus, such specifications are a poor communication medium between designers and implementors.

The SCR method is simple and practical. However, in this method, it is not always possible to describe a module's behavior with only the module's access programs. For example, in

Figure 2, we had to introduce the hidden programs *depth* and *ith* to record effects not immediately visible by using *top* alone.

Both the algebraic and trace methods provide implementation-independent specifications. They both have solid formal foundations: the former based on heterogeneous algebra and the latter on first order logic. Once a specification is constructed using either method, the consistency and completeness of the axioms in the specification can be studied mathematically [11]. Hence, design errors can be discovered at an early stage; no implementation work is wasted.

Note that in Figure 3, while we specify the *bstack*, the type *Bstack* is used in the specification. Since *Bstack* is not observable to users of *bstack*, the specification is written in terms of objects not visible to the module's users. Also, while we intended to specify the *bstack* with three access calls *push*, *pop*, and *top*, we have to include an extra call *newstack* in the specification to return an "empty stack". The other weakness of this method is that it is not powerful enough to specify many modules without introducing hidden functions [3]. For example, in Figure 3, we had to use the hidden function *depth* to specify the condition that a stack is full.

Motivated by a desire to give a complete "black-box" description of a software module, in the trace method, a module's behavior is completely specified using assertions on sequences of calls (i.e., traces). No hidden programs are needed. However, trace specifications are relatively difficult to write and to read. Therefore, the application of the trace method has been relatively limited thus far.

CHAPTER 3

Trace Methodology

In this chapter, we present Hoffman's trace methodology and illustrate the use of the method by specifying the *stack*, *unique*, *priority queue*, *traversing stack*, and *binary tree* modules.

3.1. Trace Methodology

For a simple module, such as *bstack*, a trace specification can be written in an ad hoc manner as described in Section 2.4. However, if the module to be specified is large or complex, then its specification is much longer. The assertions in the semantic section become large and unintelligible. It is then very difficult to determine whether a specification contained assertions that were mutually contradictory or whether the specification completely characterized the behavior of the module. Small changes to the desired characteristics of a module might result in large changes to the specification. To overcome these difficulties, Hoffman developed a methodology for handling complex trace specifications [1].

The first step in this methodology is writing the syntax section as before. Then, using the heuristics described below, the semantics section is written, transforming the informal notions of correct module behavior into assertions about trace legality, equivalence and value.

HEURISTIC 1 - 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 " $\equiv$ ", and the set T_L of all legal traces. It

follows from the definition of “ $\equiv$ ” (in Section 2.4) that it is an equivalence relation and so partitions T_L into a set of equivalence classes. A normal form for S contains 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. Usually a normal form is chosen which is simple to describe and which makes the next heuristic easy to apply.

HEURISTIC 2 - Structure the Semantics According to Normal Form Prefixes.

Given a normal form, module behavior for all traces is expressed in terms of module behavior for just the normal form traces. Specifically, the semantic assertions are based 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 3 - 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.

3.2. Specification Examples

3.2.1. Stack

A trace specification for an unbounded integer *stack* using the trace methodology is shown in Figure 5. The *stack* is very similar to the *bstack* in Chapter 2, except that the size of *stack* is infinite.

Figure 5. New Stack Trace Specification

```

NAME
    stack
SYNTAX
    push:  integer;
    pop:   ;
    top:           → integer;
PREDICATES
    normalform(T) ↔ length(T) = count(push,T)
SEMANTICS
    (∀ T,C) (normalform(T) →
        (∀ 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))))

```

The stack normal form is chosen to be the set of traces consisting solely of *push* calls. Although there are an infinite number of possible normal forms for this module, this choice is attractive because there is a natural correspondence between a sequence of *push* calls and our conception of the contents of the stack. 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. Note that the legality assertion for *push* states that any sequence of *push* calls is legal.

3.2.2. Unique

The *unique* module is taken from McLean [4]. The module provides unique integers for such applications as 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 Figure 6), the normal form is chosen to be all traces: in fact no other choice is possible, because all traces are legal and no two different traces are equivalent. In other words, the equivalence classes formed by ' $\equiv$ ' are all singletons. The semantics section is straightforward. *T.getint* is always legal and *getint* always returns a "new" value. No equivalence assertion is needed since *T.getint* is always normal form itself.

3.2.3. Priority Queue

The priority queue (adapted from McLean [4], hereafter *priqueue*) provides three procedures: *insert*, *remove*, and *front*. The module stores an unbounded number of ordered pairs of integers, which are inserted by *insert*. The first member of each ordered pair acts as the priority key for the second member. *front* returns (without removing) the integer with maximum prior-

Figure 6. Unique Trace Specification

```

NAME
    unique
SYNTAX
    getint:      → integer;
PREDICATES
    normalform(T) ↔ true
SEMANTICS
    (∀ T,C) (normalform(T)
              (C = getint →
                /*L*/ L(T.C) &
                /*V*/ (L(T.C) →
                        (∀ T1) (prefix(T1,T) → V(T1) ≠ V(T.C)))
                ))

```

ity. *remove* removes the pair with maximum priority.

The specification of the *priqueue* module is shown in Figure 7.

The normal form is chosen to be the set of traces consisting solely of *insert* calls, sorted by the priority key in ascending order. Then, if a normal form T is not empty, *front* returns the second member of the pair inserted by the rightmost *insert* in T , and *remove* removes the pair.

3.2.4. Traversing Stack

As in the *stack* module, the *traversing stack* module (taken from Hoffman [1], hereafter *tstack*) permits elements to be added or removed only at the top of the stack. However, *tstack* provides read access to all elements of the stack, through the calls *down* and *to_top* (see Figure 8 for the syntax of *tstack*).

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 current and *to_top* causes the element on the top of the stack to become current. *Push* and

Figure 7. Priqueue Trace Specification

NAME
 priqueue

SYNTAX
 insert: integer integer;
 remove: ;
 front: → integer;

PREDICATES
 $\text{normalform}(T) \leftrightarrow \text{length}(T) = \text{count}(\text{insert}, T) \ \& \ (\forall T1, T2, T3, p, d, p1, d1) \ (T = T1.\text{insert}(p, d).T2.\text{insert}(p1, d1).T3 \rightarrow p \leq p1)$

SEMANTICS
 $(\forall T, C)(\text{normalform}(T) \rightarrow$
 $(\forall i)(C = \text{insert}(p, i) \rightarrow$
 $/*L*/ \ L(T.C) \ \& \ (L(T.C) \rightarrow$
 $/*E*/ \ (\forall T1, T2)(T = T1.T2 \ \& \ \text{normalform}(T1.C.T2) \rightarrow T.C \equiv T1.C.T2))$
 $) \ \& \ (C = \text{remove} \rightarrow$
 $/*L*/ \ (L(T.C) \leftrightarrow T \neq e) \ \& \ (L(T.C) \rightarrow$
 $/*E*/ \ (\forall T1, p, d)(T = T1.\text{insert}(p, d) \rightarrow T.C \equiv T1))$
 $) \ \& \ (C = \text{front} \rightarrow$
 $/*L*/ \ (L(T.C) \leftrightarrow T \neq e) \ \& \ (L(T.C) \rightarrow$
 $/*E*/ \ (T.C \equiv T) \ \& \ /*V*/ \ (\forall T1, p, d)(T = T1.\text{insert}(p, d) \rightarrow V(T.C) = d)))))$

Figure 8. Traversing Stack Syntax

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

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 augmented 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 Figure 9 and makes use of the predicate *prefix*: *prefix(S,T)* is true if and only if S is a prefix of T .

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

$T1 = \text{push}(1), C1 = \text{push}(2), T2 = \text{push}(3).\text{push}(4), D = \text{down.down}$

The *parse* predicate (see Figure 9) 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 Figure 10) that is quite similar to that of *stack*.

Figure 9. Traversing Stack Predicates

PREDICATES

$$\text{normalform}(T) \leftrightarrow (\exists T1, T2) (\text{prefix}(T, T1.T2) \ \& \ \text{all}(\text{push}, T1) \ \& \ \text{all}(\text{down}, T2) \ \& \ \text{length}(T1) > \text{length}(T2))$$

$$\text{parse}(T1, C, T2, D) \leftrightarrow (T1.C.T2.D = e \mid \text{all}(\text{push}, T1.C.T2) \ \& \ \text{all}(\text{down}, D) \ \& \ \text{length}(C) \leq 1 \ \& \ \text{length}(T2) = \text{length}(D))$$

$$\text{all}(C, T) \leftrightarrow \text{count}(C, T) = \text{length}(T)$$

3.2.5. Binary Tree

In this section, we present a fairly complex module: the *binary tree* module (hereafter *tree*).

We describe the basic task, motivate the need for a precise specification, and present a trace specification.

Figure 10. Traversing Stack Semantics

SEMANTICS

$$\begin{aligned} &(\forall T, C) (\text{normalform}(T) \rightarrow \\ &(\forall T1, C1, T2, D) (T = T1.C1.T2.D \ \& \ \text{parse}(T1, C1, T2, D) \rightarrow \\ &(\forall i)(C = \text{push}(i) \rightarrow \\ & \quad /*L*/ \ L(T.C) \leftrightarrow D = e) \ \& \\ & \quad (C = \text{pop} \rightarrow \\ & \quad /*L*/ \ (L(T.C) \leftrightarrow D = e \ \& \ C1 \neq e) \ \& \\ & \quad /*E*/ \ (L(T.C) \rightarrow T.C \equiv T1)) \ \& \\ & \quad (C = \text{down} \rightarrow \\ & \quad /*L*/ \ L(T.C) \leftrightarrow \text{normalform}(T.C)) \ \& \\ & \quad (C = \text{current} \rightarrow \\ & \quad /*L*/ \ (L(T.C) \leftrightarrow C1 \neq e) \ \& \\ & \quad \quad L(T.C) \rightarrow (\\ & \quad \quad /*E*/ \quad T.C \equiv T \ \& \\ & \quad \quad /*V*/ \quad (\forall i) (C1 = \text{push}(i) \rightarrow V(T.C) = i))) \ \& \\ & \quad (C = \text{to_top} \rightarrow \\ & \quad /*L*/ \ L(T.C) \ \& \\ & \quad /*E*/ \ (L(T.C) \rightarrow T.C \equiv T1.C1.T2)) \\ & \quad)) \end{aligned}$$

Prose Specifications and Unanswered Questions

We desire a *tree* module which provides access to a binary tree. For each node in the tree, there is a value field which is either undefined or contains an integer value. We need the ability to add and delete nodes, and to determine whether a given node has a left child, right child, or parent.

There are many conceivable modules which satisfy these needs, but differ in the way they deal with numerous questions, such as

- (1) Is there a limit on the number of nodes in the tree?
- (2) If a node already has a value, can it be changed?
- (3) If a node *x* is deleted, should the sub-tree which has *x* as its root be deleted as well, or should deletion of such a node be prohibited?
- (4) If a node *x* has a left child, what should happen if we attempt to create a left child of *x*?
- (5) Is the root node of the tree treated differently from other nodes in the tree?

The answers to these and similar questions constitute design decisions, typically based on implementation issues and the intended application of the module. The goal of a *tree* specification is to precisely record these decisions, whatever they may be. We have found prose unsuitable to keeping such a record.

A Tree Trace Specification

In this section, we present a trace specification for the *tree* module. While the specification provides answers to the five questions above, the design decisions implied are to some extent arbitrary. Figure 11 contains the *tree* specification syntax (see the Appendix for

the full specification).

At any time one node in the tree is *current* and all the *tree* access calls depend on the current node. We assume that before the tree module begins execution, the root node already exists and is the current node. *setc* assigns the current node a new value and *getc* returns this value. *exp*, *exl*, and *exr* are true if and only if the current node has a parent, left child and right child, respectively. *mvp*, *mvl*, and *mvr* cause the parent, left child and right child, respectively, of the current node to become current. *mkl* (*mkr*) creates a left (right) child node of the current node, and *del* deletes the current node and all of its descendants, if any.

The *tree* module is considerably more complex than the stack module. As a result, the *tree* specification depends heavily on the heuristics of Section 3.1. Following **HEURISTIC 1**, our specification is based on a normal form. Since this normal form is much more complex than that of *stack*, we follow **HEURISTIC 3** and decompose the *normalform* predicate into

Figure 11. Tree Syntax

SYNTAX

```

setc: integer;
getc: → integer;

exl: → boolean;
exr: → boolean;
exp: → boolean;

mvl: ;
mvr: ;
mvp: ;

mkl: ;
mkr: ;
del: ;

```

simpler predicates. The most important predicate is *tree* (see Figure 12), which characterizes a set of traces, each corresponding to a binary tree listed in pre-order.

For a trace T , $tree(T)$ is true if and only if T consists of sub-traces Trt , which assigns a value to the root (the current node); Tl , which creates a left sub-tree, and Tr , which creates a right sub-tree. $Tl(Tr)$ is either empty or consists of a *mk* (*mk*) call which creates the root of the left (right) sub-tree, a sub-trace Tl which creates the sub-tree in pre-order and makes its root the current node, and a *mv* call which makes the root of $Tl(Tr)$ the current node.

The traces which satisfy *tree* correspond to the contents of a tree. A normal form trace also has a sequence of *mv* and *mvr* calls corresponding to the current position in the tree. We say that $T1.Tc.T2$ matches Tm if the *mv* and *mvr* calls in Tm correspond to a path in the tree of $T1.Tc.T2$ beginning at the root.

We write a predicate for the normal form and also several “utility” predicates (see Figure 13).

The *parse* predicate splits a normal form trace into the following four parts:

Tc : the subtrace that creates the current sub-tree;

Figure 12. Predicates Specifying A Tree

PREDICATES

$$tree(T) \leftrightarrow (\exists Trt, Tl, Tr) \\ (T = Trt.Tl.Tr \ \& \ root(Trt) \ \& \ ltree(Tl) \ \& \ rtree(Tr))$$

$$root(T) \leftrightarrow (T = e \mid (\exists i)(T = setc(i)))$$

$$ltree(T) \leftrightarrow (T = e \mid (\exists T1)(T = mkl.T1.mvp \ \& \ tree(T1)))$$

$$rtree(T) \leftrightarrow (T = e \mid (\exists T1)(T = mkr.T1.mvp \ \& \ tree(T1)))$$

Figure 13. Tree Normal Form Predicate

PREDICATES

$$\text{normalform}(T) \leftrightarrow (\exists T1, Tc, T2, Tm)(T = T1.Tc.T2.Tm \ \& \ \text{parse}(T1, Tc, T2, Tm))$$

$$\text{parse}(T1, Tc, T2, Tm) \leftrightarrow (\text{tree}(T1.Tc.T2) \ \& \ \text{match}(T1.Tc.T2, Tm, Tc) \ \& \ \text{allmv}(Tm))$$

$$\begin{aligned} \text{match}(Tt, Tm, Tc) \leftrightarrow & (\text{tree}(Tt) \ \& \ \text{tree}(Tc) \ \& \ \text{allmv}(Tm) \ \& \\ & (Tm = e \ \& \ Tt = Tc \mid \\ & (\forall C, Tail, Trt, Tl, Tr) \\ & (Tt = Trt.Tl.Tr \ \& \ \text{root}(Trt) \ \& \ \text{ltree}(Tl) \ \& \\ & \text{rtree}(Tr) \ \& \ Tm = C.Tail \rightarrow \\ & ((C = mvl \rightarrow (\exists Tl1) \\ & \quad (Tl = mkl.Tl1.mvp \ \& \ \text{match}(Tl1, Tail, Tc))) \ \& \\ & (C = mvr \rightarrow (\exists Tr1) \\ & \quad (Tr = mkr.Tr1.mvp \ \& \ \text{match}(Tr1, Tail, Tc)))))) \\ &))) \end{aligned}$$

$$\text{allmv}(Tm) \leftrightarrow (\text{count}(mvl, Tm) + \text{count}(mvr, Tm) = \text{length}(Tm))$$

$T1$ and $T2$: such that $\text{tree}(T1.Tc.T2)$ is true; and

Tm : consisting solely mvl and mvr calls corresponding to a path from the root to the current node.

For the tree example given in Figure 14, assuming that its current node is the node that has value 4 and that $\text{normalform}(T)$ is true, we have

$$\begin{aligned} T &= \text{setc}(1).mkl.\text{setc}(2).mkl.\text{setc}(4).mkl.\text{setc}(5).mvp. \\ &\quad mkr.\text{setc}(6).mvp.mvp.mvp.mkr.\text{setc}(3).mvp.mvl.mvl \\ T1 &= \text{setc}(1).mkl.\text{setc}(2).mkl \\ Tc &= \text{setc}(4).mkl.\text{setc}(5).mvp.mkr.\text{setc}(6).mvp \\ T2 &=.mvp.mvp.mkr.\text{setc}(3).mvp \\ Tm &= mvl.mvl \end{aligned}$$

For the most part, the semantics present straightforward answers to the questions posed at the beginning of this section. We only examine *del* in detail (see Figure 15). Informally, the *del* semantics are as follows. The legality section says that *del* is legal if the tree is not empty.

Figure 14. A Tree Example

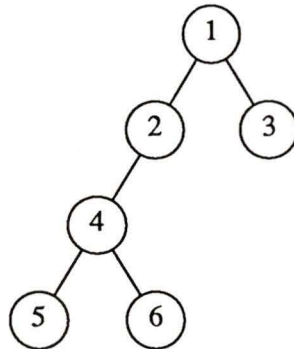


Figure 15. Del Semantics

SEMANTICS

```

(∀ T,C)(normalform(T) →
(∀ T1,Tc,T2,Tm)(T=T1.Tc.T2.Tm & parse(T1,Tc,T2,Tm) →
(∀ Tcrt,Tcl,Tcr)(Tc=Tcrt.Tcl.Tcr & tree(Tc) &
  root(Tcrt) & ltree(Tcl) & rtree(Tcr) →
  ... ..

(C=del →
/*L*/ (L(T.C) ← T≠e) &
/*E*/ (L(T.C) →
  ((Tm=e → T.C≡e) &
  (∀ T11,T22,Tm1)
    (T1=T11.mkl & T2=mvp.T22 & Tm=Tm1.mvl |
    T1=T11.mkr & T2=mvp.T22 & Tm=Tm1.mvr →
    T.C≡T11.T22.Tm1))))
)))

```

The equivalence section states that if the current node is the root of the tree ($Tm=e$), then only the root node will remain. Otherwise, the current sub-tree (Tc) is deleted from the tree.

CHAPTER 4

Prototype Building

In this chapter, we present our method for constructing Prolog prototypes from trace specifications, and illustrate it on the *stack*, *traversing stack*, and *tree* modules. To distinguish between trace and Prolog objects, throughout the chapter, elements in the trace language are written in *italics* while their denotations in Prolog are written in typewriter-style.

4.1. Scheme

The trace methodology (see Section 3.1) provides a method for writing specifications for modules in a uniform way. The prototype development technique we choose is also uniform across modules. We represent a trace as a (possibly empty) list of calls, where a call is itself a 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* is represented as `[push(1), push(2), pop, top]`.

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 Prolog prototypes by writing procedures `lp`, `nf`, and `val`, each taking T and C as parameters. $lp(T, C)$ is satisfiable if and only if $T.C$ is legal. $nf(T, C, Nftr)$ is satisfiable if and only if $Nftr$ is a normal form trace equivalent to $T.C$ and $val(T, C, V)$ is satisfiable if and only if V is $V(T.C)$. For the most part, we generate prototypes by translating the specification assertions, line by line, from predicate calculus to Prolog.

One problem remains: `lp`, `nf`, and `val` operate only on normal form traces, while prototypes must also handle traces not in normal form. The solution is provided by procedures `l` and `v`. For any trace T , `l(T)` is satisfiable if and only if $L(T)$ is true and `v(T, i)` is satisfiable if and only if $V(T) = i$. The procedure `nfdriver` recursively applies `lp` and `nf` to detect an illegal trace and transform a legal one into its normal form equivalent.

The procedures `l`, `v` and `nfdriver` (see Figure 16) are the same (module independent) for all prototypes, while `lp`, `nf` and `val` are different for each prototype.

There are three situations in which the direct trace-to-Prolog translation fails: (1) The specification contains constructions, such as “ $\exists$ ”, with no direct Prolog equivalent. (2) Due to Prolog’s depth-first search scheme, a correct direct translation does not terminate. (3) The specification is non-deterministic, i.e., for some function call, the specification allows two or more different values to be returned. Though we do not have a general scheme for solving these problems, in our prototype building experience, the problems have been simple to work around. In cases (1) and (2), we have generated prototype code by slightly restructuring the specification assertions. In case (3), we have added constraints so the Prolog code chooses one return value from those permitted by the specification, making the prototype deterministic.

Figure 16. Module Independent Procedures

```

l(T) :- nfdriver([], T, Nftr) .

v(T, V) :- append(T1, C, T), nfdriver([], T1, Nftr), val(Nftr, C, V) .

nfdriver(Nftr, [], Nftr) .
nfdriver(Nf, [C|T1], Nftr) :- lp(Nf, [C]),
    nf(Nf, [C], Nf1), nfdriver(Nf1, T1, Nftr) .

```

4.2. Prototype Examples

4.2.1. A Stack Prototype

The `lp`, `nf` and `val` procedures for the *stack* prototype (shown in Figure 17) are derived directly from the specification of Figure 5. `lp` is generated from the lines dealing with legality, `nf` is generated from the lines dealing with equivalence, and `val` is generated from the lines dealing with value. Since *push* and *pop* do not return values, `val` contains a line for `top` only.

If we execute

```
l([push(1),push(2),pop,top])
```

then the `nfdriver` sub-goals will be

```
nfdriver([], [push(1),push(2),pop,top], Nftr)
nfdriver([push(1)], [push(2),pop,top], Nftr)
```

Figure 17. Stack Prototype

```
normalform(T) :- length(T,I), count(push,T,I).

count(push,[],0).
count(push,[push(_) | T],J) :- count(push,T,I), J is I+1.
count(push,[C | T],J) :- (C = pop | C = top), count(push,T,J).

lp(T,[push(_) ]).
lp(T,[top]) :- append(_, [push(_) ],T).
lp(T,[pop]) :- append(_, [push(_) ],T).

nf(T,[push(S)],Nftr) :- append(T,[push(S)],Nftr).
nf(T,[top],Nftr) :- append([],T,Nftr).
nf(T,[pop],Nftr) :- append(Nftr,[push(_) ],T).

val(T,[top],V) :- append(_, [push(V) ],T).
```

```

nfdriver([push(1),push(2)], [pop,top], Nftr)
nfdriver([push(1)], [top], Nftr)
nfdriver([push(1)], [], [push(1)])

```

and the `nfdriver` sub-goal in procedure 1 is satisfiable, indicating that *push(1).push(2).pop.top* is a legal trace. If, however, we execute

```
1([push(1),push(2),pop,pop,top])
```

then the `nfdriver` sub-goals will be

```

nfdriver([], [push(1),push(2),pop,pop,top], Nftr)
nfdriver([push(1)], [push(2),pop,pop,top], Nftr)
nfdriver([push(1),push(2)], [pop,pop,top], Nftr)
nfdriver([push(1)], [pop,top], Nftr)
nfdriver([], [top], Nftr)

```

To satisfy the last `nfdriver` sub-goal, `lp([],top)` must be satisfied. But, `lp([],top)` is unsatisfiable. Thus, `nfdriver([], [top], Nftr)` is also unsatisfiable. Backtracking to 1,

```
1([push(1),push(2),pop,pop,top])
```

is unsatisfiable, indicating that *push(1).push(2).pop.pop.top* is an illegal trace.

Procedure `v` is handled in a similar way, except that only the head of the trace (the trace without the last call) is passed to `nfdriver` and then `Nftr` and the last function call are passed to `val`. For example, `v([push(1),push(2),pop,top], V)` instantiates `V` as 1.

4.2.2. A Traversing Stack Prototype

The *stack* and *tstack* specifications are quite similar except for the presence of the *parse* predicate in the latter. Appropriately, the two prototypes are similar except for the *parse* pro-

cedure (shown in Figure 18) in the *tstack* prototype.

In the specification, the *parse* predicate is used to assert that if a trace *T* is divided into the substraces *T1*, *C1*, *T2* and *D*, then certain assertions hold of the substraces. In the prototype, the *parse* procedure takes the parameter *Nftr*, and finds the four substraces, so that new traces can be constructed from them.

There are two sub-goals in procedure *parse*: *parse1* and *parse2*. *parse1*(*T*, *P*, *D*) is satisfiable if and only if *T* is the concatenation of two substraces *P* and *D*, such that *P* contains only *push* calls and *D* contains only *down* calls (both *P* and *D* may be the empty trace). *parse2*(*T*, *D*, *T1*, *T2*, *C1*) is satisfiable if and only if *T* is the concatenation of *T1*, *C1*, *T2* and *D* such that

Figure 18. Parse Procedures

```

parse([], [], [], [], []).
parse(T, T1, C, T2, D) :- parse1(T, P, D), parse2(P, D, T1, C, T2).

parse1(T, P, D) :- append(P, D, T), alld(D), allp(P).

parse2(P, D, T1, [C], T2) :-
    length(P, L1), length(D, L2), L1 > L2,
    subtrace(P, L1-L2, Temp1, T2), append(T1, [C], Temp1).

subtrace(T, 0, [], T).
subtrace([C|T], I, [C|T1], T2) :-
    I > 0, length(T, L), I <= L+1,
    J is I-1, subtrace(T, J, T1, T2).

alld([]).
alld([down|T]) :- alld(T).

allp([]).
allp([push_|T]) :- allp(T).

```


(1) $T1$, $C1$ and $T2$ contain only *push* calls (both $T1$ and $T2$ may be empty trace);

(2) D contains only *down* calls (may be empty);

(3) $length(T2) = length(D)$; and

(4) $C1$ is a *push* call.

Clearly, if $parse(T, T1, C1, T2, D)$ is satisfiable, then T is a normal form trace. With $parse$ available, the lp , nf , and val procedures are coded directly from the specification (see Figure 19). Essentially, the *tstack* prototype is just the *stack* prototype with calls to the $parse$ procedure added.

4.2.3. A Binary Tree Prototype

Although *tree* is considerably more complex than both *stack* and *tstack*, our technique for generating a Prolog prototype still works well. For the predicates describing a tree, the Prolog

Figure 19. Traversing Stack Prototype

```
normalform(T) :- parse(T,_,_,_,_).

lp(T,[push(_)]) :- parse(T,_,_,_,[]).
lp(T,[pop]) :- parse(T,_,[push(_)],_,[]).
lp(T,[down]) :- append(T,[down],Temp), normalform(Temp).
lp(T,[current]) :- parse(T,_,[push(_)],_,_).
lp(T,[to_top]).

nf(T,[push(X)],Nftr) :- append(T,[push(X)],Nftr).
nf(T,[pop],Nftr) :- append(Nftr,[push(_)],T).
nf(T,[down],Nftr) :- append(T,[down],Nftr).
nf(T,[current],Nftr) :- append(T,[],Nftr).
nf(T,[to_top],Nftr) :- parse1(T,Nftr,_).

val(T,[current],V) :- parse(T,T1,[push(V)],T2,D).
```

procedures are shown in Figure 20. There is a clear correspondence between the predicates and Prolog procedures.

For the most part, the Prolog procedures are straightforward. Here we describe only the procedures that deal with the *del* call, shown in Figure 21. (The complete Prolog prototype is in the Appendix). *parse* corresponds to the predicate *parse* in the specification. *treeparse*(*T*, *Trt*, *Tl*, *Tr*) is satisfiable if *Trt* corresponds to the root of *T*, *Tl* and *Tr* correspond the left and right sub-tree of *T*, and *lp*(*T*, [*del*]) is satisfiable if the normal form trace *T* is not empty. If *normalform*(*T*) is true and *Tc* corresponds the current sub-tree, then *nf*(*T*, [*del*], *Nftr*) is satisfiable if *Nftr* is *T* with *Tc* removed. In Figure 21, *nf* is broken into three cases, depending whether the current node is a left child, a right child, or is not a child (i.e., is the root of the entire tree).

4.3. Prototype User Interfaces

In this section, we present two kinds of prototype user interface that will make the Prolog prototypes much easier to use: the Prolog interface and the C [15,16] interface.

Figure 20. Tree Prolog Procedures

```
tree([]).
tree(T) :- append(Tl,Tr,T), ltree(Tl), rtree(Tr).
tree([setc(_) | T]) :-
    append(Tl,Tr,T), ltree(Tl), rtree(Tr).

ltree([]).
ltree(T) :- append([mk1 | Tl], [mvp], T), tree(Tl).

rtree([]).
rtree(T) :- append([mkr | Tl], [mvp], T), tree(Tl).
```

Figure 21. Del Prolog Procedure

```

lp(T,[del]) :- T=[setc(I)|_]; T=[mkl|_];T=[mkr|_].

nf(T,[del],Nftr) :- append(Tt,[mvl],T),
                    parse(Tt,Tl,Tc,T2,Tm),
                    treeparse(Tc,Trt,Tl,Tr),
                    append3(Tl,Trt,Tr,Temp),
                    append3(Temp,T2,Tm,Nftr).

nf(T,[del],Nftr) :- append(Tt,[mvr],T),
                    parse(Tt,Tl,Tc,T2,Tm),
                    treeparse(Tc,Trt,Tl,Tr),
                    append3(Tl,Trt,Tl,Temp),
                    append3(Temp,T2,Tm,Nftr).

nf(T,[del],Nftr) :- append(_,[mvp],T), Nftr=[].

```

4.3.1. The Prolog Interface

The Prolog prototypes are attractive because they closely follow the specification, thereby simplifying both the coding task and the correctness arguments. However, they offer an interface that is not convenient to use. For example, in the *stack* Prolog code, even if we found that `l([push(1),pop,pop,push(2)])` is unsatisfiable, we still do not know which call makes the trace illegal. We might have to try a number of prefixes of the trace to find out the first illegal call in the trace.

To supply a more convenient interface, we provide a Prolog procedure corresponding to each call in the specification syntax section. We use the procedure `nftr` and the Prolog built-in procedures `asserta` and `retract` to store and access the current normal form trace. For the *stack* interface (see Figure 22), for example, `init` asserts `nftr([])` and each `push` and `pop` procedure first retracts `nftr(_)` and then asserts `nftr(Nftr)`, where `Nftr` is the

new normal form trace.

`init` must be called first, to initialize `nftr`. Both the `push` and `pop` procedures are satisfiable if `applycall` is satisfiable. The `applycall(C)` is satisfiable if `C` is legal (as determined by `lp`) and `nf` is satisfiable. A `top` procedure is satisfiable if the call is legal (as determined by `lp`) and `write` gives the appropriate value (using `val`).

With this Prolog interface, we may test the *stack* prototype by simply issuing each single call in a trace *T* to see if it is legal or to determine its return value. For example, to find that if the trace *push(1).pop.pop* is illegal, we type:

```
init
    which is satisfiable, followed by
push(1)
    which is satisfiable, followed by
pop
    which is satisfiable, followed by
pop
    which is unsatisfiable, indicating that push(1).pop.pop is illegal
```

In this way, prototype testing is much easier.

Figure 22. Stack Prolog Interface

```
init :- asserta(nftr([])).
push(I) :- applycall([push(I)]).
pop :- applycall([pop]).
top :- returnval([top]).
applycall(C) :- nftr(T), lp(T,C), nf(T,C,Nftr),
    retract(nftr(_)), asserta(nftr(Nftr)), !.
applycall(C) :- write('Illegal call').
returnval(C) :- nftr(T), lp(T,C),
    val(T,C,V), write(V), !.
returnval(C) :-
    write('Illegal call').
```


The `init`, `applycall`, and `returnval` procedures are the same for all prototypes. The remaining procedures are directly based on the syntax part of that module's trace specification. We have developed software to automatically generate an interface for a prototype from the syntax part of the module's specification.

4.3.2. The C Interface

Thus far we've presented two interfaces for access to executable prototypes: by way of the `l`, `v` calls or, more conveniently, through the calls in the specification syntax. In this section, we show how to access a prototype from C language code, so that the prototype may serve as part of a larger system, at least until a more efficient replacement can be written.

Because the C and Prolog run time environments are incompatible, we use two processes: a C application process and a Prolog process. The interface is based on the `prog` module coded in C and providing three calls:

```
prog_open(pgmname)
```

creates *pgmname* as a separate process, with *virtual terminals* and *pseudo-teletypes* [16] supporting two-way communication through *pgmname*'s standard input and standard output.

```
prog_getc
```

returns characters sent by *pgmname* to its standard output, one at a time.

```
prog_putc(c)
```

sends *c* to *pgmname*'s standard input.

The application process is coded entirely in C and consists of three modules:

- (1) The application code includes code which calls `init`, and the access programs of the prototype (e.g., for *stack*, `push`, `pop`, and `top`).
- (2) The C interface implements `init` with a call to `prog_open`. The access programs are implemented by using `prog_putc` and `prog_getc` to pass strings between the C code and the Prolog prototype.
- (3) The `prog` module is described above.

The Prolog process also consists of three modules: (4) the Prolog prototype interface as described in Section 4.3.1, (5) the Prolog prototype, and (6) the Prolog interpreter. Most of the difficulty in implementing the C interface is in module (3). To adapt the interface to a different prototype requires only simple changes to modules (2) and (4).

CHAPTER 5

Prototype Correctness Proof

The prototype building method described in Chapter 4 is straightforward. We are confident of the correctness of the prototypes thus developed. However, it is desirable that we can prove the correctness of the prototypes. We do the proof by viewing our prototypes as trace models. In [4], McLean defines trace model and model correctness. In Section 5.1, we present McLean's model definition with his stack model example. Then, in Section 5.2, we show how our Prolog prototypes can be viewed as McLean models and present a scheme for proving that such a model meets its specification. In 5.3, we use the proof scheme to prove the correctness of our stack prototype.

To distinguish between trace and model objects, in this chapter, elements in the trace language are written in *italics* while their denotations in Prolog are written in typewriter-style.

For simplicity, throughout the chapter, the following two notations are used:

- (1) For a Prolog procedure *proc*, we introduce two boolean expressions: *sat* and *unsat*. *sat(proc)* is true if and only if *proc* is satisfiable. *unsat(proc)* is true if and only if *sat(proc)* is false.
- (2) We write *L1.L2* to indicate the list *L* for which *sat(append(L1, L2, L))* is true. For example, rather than writing *append(L1, L2, L)* and *P(L)* to express that *L* satisfies procedure *P* and is the concatenation of *L1* and *L2*, we instead write *P(L1.L2)*.

Throughout the chapter, we assume that all traces are syntactically correct, that T and Nf are trace constants, and that $Nftr$ is a trace variable.

5.1. McLean's Model

In this section, we present McLean's model definition with a stack example. The correctness of the stack model is also briefly discussed.

5.1.1. Model Definition

Adapted from McLean

A *trace model* is an ordered pair (D, I) where D is a tuple of domains and I is a function from syntactic constructs in the trace language to their denotations in D .

$D = (D_T, D_e, D_L, D_{Vint}, D_{Vbool}, D_{INT}, D_{BOOL})$ and $D_T \supseteq D_L \supseteq D_e$, $D_T \supseteq D_{Vint}$, $D_T \supseteq D_{Vbool}$. The constraints on the function I are described in Figure 23.

Intuitively, D_T contains denotations for traces. D_e contains the denotation of the empty trace. D_L contains denotations for legal traces. D_{Vint} and D_{Vbool} contain denotations for traces that end in a function call of type integer and boolean respectively. D_{BOOL} is a return value domain containing denotations for boolean constants and D_{INT} is the parameter and return value domain containing denotations for integer constants.

5.1.2. Stack Model

We present a model for the stack specification shown in Figure 24, adapted from McLean [4]. The stack module is the same as specified in Figure 5, however, the specification is in the old trace style.

Figure 23. Requirements for Function I

- (1) $I[=] = \{(X, X) : X \in \bigcup D\}$
- (2) $I[L] = D_L$
- (3) $I[V] = h : (D_{Vint} \cup D_{Vbool}) \cap D_L \rightarrow (D_{INT} \cup D_{BOOL})$ such that for $T \in D_T$
 $h(T) \in D_{INT}$ if $T \in D_{Vint} \cap D_L$ and
 $h(T) \in D_{BOOL}$ if $T \in D_{Vbool} \cap D_L$
- (4) $I[.] = f : D_T \times D_T \rightarrow D_T$ such that for all $X, Y, Z \in D_T$
 (a) $f(f(X, Y), Z) = f(X, f(Y, Z))$
 (b) $f(X, Y) = X$ if $Y \in D_e$
 (c) $f(X, Y) = Y$ if $X \in D_e$
 (d) $f(X, Y) \in D_{Vint}$ iff $[Y \in D_{Vint} \text{ or } (Y \in D_e \text{ and } X \in D_{Vint})]$ and
 $f(X, Y) \in D_{Vbool}$ iff $[Y \in D_{Vbool} \text{ or } (Y \in D_e \text{ and } X \in D_{Vbool})]$
 (e) $f(X, Y) \in D_T - D_L$ if $X \notin D_L$
- (5) $I[f] = g : I[\delta_1] \times I[\delta_2] \dots \times I[\delta_n] \rightarrow I[\delta^*]$
 where domain function $f : \delta_1 \times \delta_2 \dots \times \delta_n \rightarrow \delta^*$.
- (6) $I[R] \subseteq I[\delta_1] \times I[\delta_2] \dots \times I[\delta_n]$
 where R is a n -ary domain relation on domain $\delta_1 \times \delta_2 \dots \times \delta_n$.
- (7) For trace expression
 (a) $I[e] \in D_e$
 (b) $I[\phi] \in D_T$ for any trace variable ϕ
 (c) $I[\phi(\alpha_1, \dots, \alpha_n)] = I[\phi](I[\alpha_1], \dots, I[\alpha_n])$
 (d) $I[T.R] = I[.](I[T], I[R])$

In this model, $D_T = \text{TRACE}$, the set of (possibly null) character strings that form sequences of procedure calls, contains denotations for traces; $D_{INT} = \text{INT}$, the set of integers, contains denotations for integer constants. For example, the string "push(6).top.pop" is a TRACE domain element corresponding to the trace *push(6).top.pop*.

D_L is a subset of TRACE . A string $x \in D_L$ if and only if, in x , for any substring ϕ of the form "pop" or "top", to the left of ϕ , there are more substrings of the form "push" than of the form "pop". D_{Vint} contains denotations for traces that end in a *top* call. $I[V]$ is a function from $D_L \cap D_{Vint}$ to integers. For $x \in D_L \cap D_{Vint}$, $f(x) = n$ if and only if, scanning x from right to left, when all the "push-pop" pairs and "top" substrings are deleted from

Figure 24. A Stack Trace Specification

```

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)

```

x , the rightmost substring in x is “push (n)”.

5.1.3. The Stack Model's Correctness

Since McLean's stack model is expressed in pseudocode, its correctness proof must be informal. To show the correctness of the stack model we need to show that

- (1) the stack model meets McLean's requirements, and
- (2) every assertion in the specification (see Figure 24) is true in the model

Since the model is simple and the elements in the TRACE domain correspond in the obvious way to traces, the proof is simple (but tedious). Here, we only show that the model meets requirements (1) - (3) (see Figure 23) and that assertion (1) in the specification (see Figure 24) is true in the model.

It is clear that in the stack model, $I[=]$ is the standard equality. Thus, for any element $x \in D$, which is either a string or an integer, we have $x = x$. The model meets requirement (1). For requirement (2), by definition, we have $D_L = I[L]$. Then, it is clear that $I[V]$ is a function from $D_{Vint} \cap D_L$ to INT , i.e., the model meets requirement (3).

Now, we prove that the assertion $(\forall T, i)(L(T) \rightarrow L(T.push(i)))$ is true in the model. Suppose $(I[L])(I(T))$. Since $I(T.push(i))$ differs from $I(T)$ only by having an additional "push(i)" substring at its end, the number of "push"s and the number of "pop"s to the left of any "top" or "pop" substring in $I(T.push(i))$ are the same as those in $I(T)$. Thus $(I[L])(I(T.push(i)))$. Therefore, the assertion holds in the model.

5.2. Our Model

In our Prolog prototypes, we have

$$D = (D_T, D_e, D_L, D_{Vint}, D_{Vbool}, D_{INT})$$

where D_T is the trace domain and each of its elements is a Prolog list. $D_e = \{ [] \} \subseteq D_T$. $D_L = \{ T \mid \text{sat}(l(T)) \}$. $D_{Vint} = \{ T \mid T \text{ ends in a function call of type integer} \}$. $D_{Vbool} = \{ T \mid T \text{ ends in a function call of type boolean} \}$. D_{INT} is the set of integers, containing denotations for integer and boolean constants ($I[true] = 1$ and $I[false] = 0$). It is both a return value and parameter domain.

We base our definition of $I[L]$ and $I[V]$ on l and v (see Figure 16) respectively.

Thus, we have

$$\begin{aligned} I[L] &= \{ T \mid \text{sat}(l(T)) \} \quad \text{and} \\ I[V](T) &= i \text{ if and only if } \text{sat}(v(T, i)). \end{aligned}$$

$I[.]$ is defined by the Prolog procedure `append`.

5.2.1. Proof Scheme

In proving the correctness of a Prolog prototype with respect to its specification, we follow the following steps:

- Prove that the prototype is a McLean model.
 - (1) Prove that l terminates.
 - (2) Prove that v terminates and is functional.
 - (3) Prove that the prototype meets McLean's requirements (5) and (6).
- Prove that the prototype is a model of the specification.
 - (4) Prove that $\text{sat}(lp)$ and $\text{sat}(nf)$ in order to show that the *Idempotency Lemma* (see Section 5.2.3) applies.
 - (5) Prove that every assertion in the specification is true in the model.

5.2.2. Prolog Prototype and Trace Model

Termination Problem

Since we have defined

$$I[L] = \{ T \mid \text{sat}(l(T)) \} \quad \text{and} \\ I[V](T) = \perp \text{ if and only if } \text{sat}(v(T, \perp)).$$

It is clear that for $I[L]$ and $I[V]$ to be well defined, l and v must terminate. From Figure 16, we have that $l(T)$ terminates if and only if $\text{nfdriver}([], T, \text{Nftr})$ does; and $v(T, V)$ terminates if and only if both $\text{nfdriver}([], T, \text{Nftr})$ and $\text{val}(\text{Nftr}, C, V)$ do. Therefore, the key step in proving that l and v terminate is to prove that nfdriver terminates.

Since `lp`, `nf`, and `val` are model dependent, we have to separately prove their termination for each individual model. Then, assuming that `lp`, `nf`, and `val` terminate, by the following *Termination Lemma* and its corollary, we can prove that `nfdriver` terminates. We also assume that `append` terminates if any one of its parameters is instantiated.

Termination Lemma: For any traces Nf and T , if for each single call C in T , $lp(Nf, [C])$ and $nf(Nf, [C], Nftr)$ terminate, then $nfdriver(Nf, T, Nftr)$ terminates.

Proof: Our proof is by induction on the length of T . For the basis step, we know that $nfdriver(Nf, T, Nftr)$ terminates if the length of T is 0, i.e., $T = e$, since $nfdriver(Nftr, [], Nftr)$ is a fact.

For the induction hypothesis, suppose that $nfdriver(Nf, T, Nftr)$ terminates on any trace Nf and T of length less than n . For a trace T of length n , $nfdriver(Nf, [C1|T1], Nftr)$ (where $C1.T1 = T$) terminates if $lp(Nf, [C1])$, $nf(Nf, [C1], Nf1)$ and $nfdriver(Nf1, [C1], Nftr)$ terminate.

Then, since $T1$ is of length $n-1$, by the induction hypothesis, $nfdriver(Nf1, T1, Nftr)$ terminates. Thus, $nfdriver(Nf, T, Nftr)$ terminates. $\square$

Corollary: For any trace T , if for each single call C in T , $lp(Nf, [C])$ and $nf(Nf, [C], Nftr)$ terminate then $nfdriver([], T, Nftr)$ terminates.

Proof: This is a special case of the *Termination Lemma*, i.e., $Nf = e$. $\square$

From Figure 16, we have

$l(T) :- nfdriver([], T, Nftr)$

By this *Corollary*, we already proved that l terminates. Then, for each prototype, we need to

prove that `val` terminates in order to prove that `v` terminates.

Trace Model Requirements

We now show that our choice for the function `I` meets McLean's requirements shown in Figure 23.

Requirement (1): in Prolog, for each $X \in D$, $X = X$ holds. Since $D_L = \{ T \mid \text{sat}(1(T)) \}$, we have $I[L] = D_L$. Thus, requirement (2) is met. For requirement (3), in our prototypes, $I[V]$ is defined by the procedure `v`:

`v(T,V) :- append(T1,C,T), nfdriver([],T1,Nftr), val(Nftr,C,V).`

If `v(T,V)` is satisfiable, then $T \in D_L$ and $C \in D_{V_{int}}$ (or $D_{V_{bool}}$). If `val` is satisfiable, V is instantiated to an element $x \in D_{INT}$. The procedure `val` is module dependent. If it is functional, i.e., for any given normal form trace `Nftr` and a function call `C`, there exists only one V such that $\text{sat}(\text{val}(\text{Nftr}, C, V))$, then it is clear that `v` is also functional. Thus, for each Prolog prototype, if we can prove that `val` is functional, then it meets requirement (3).

To show that our prototypes meet requirement (4), we have to show that, $I[.] = f: D_T \times D_T \rightarrow D_T$ such that for all $X, Y, Z \in D_T$

$$(a) \quad f(f(X, Y), Z) = f(X, f(Y, Z))$$

In our prototypes, $I[.]$ is implemented by `append`, which is associative.

$$(b) \quad f(X, Y) = X, \text{ if } Y \in D_e$$

When $Y \in D_e$, i.e., $Y = []$, it is obvious that `append(X, [], T)` is satisfiable if and only if T is instantiated as X .

$$(c) \quad f(X, Y) = Y \text{ if } X \in D_e$$

When $X \in D_e$, i.e., $X = []$, it is obvious that `append([], Y, T)` is satisfiable

if and only if T is instantiated as Y .

(d) $f(X, Y) \in D_T - D_{Vint}$, if $Y \notin D_{Vint} \cup D_e$ and

$f(X, Y) \in D_T - D_{Vbool}$, if $Y \notin D_{Vbool} \cup D_e$

First, it is true that if $\text{append}(X, Y, T)$ is satisfiable, then $T \in D_T$. Assume that $Y \notin D_{Vint} \cup D_e$. Since $Y \notin D_{Vint}$, the last call in Y can't be a function call. Further, $Y \notin D_e$, i.e., $Y \neq []$. Thus, it is guaranteed that T does not end in a function call, that is $T \notin D_{Vint}$. The rest of the proof is very similar.

(e) $f(X, Y) \in D_{Vint}$ if $Y \in D_{Vint}$ and

$f(X, Y) \in D_{Vbool}$ if $Y \in D_{Vbool}$

It is clear that if $\text{append}(X, Y, T)$ is satisfiable, the last call of Y is also the last call of T . Then it is clear that (e) holds.

(f) $f(X, Y) \in D_T - D_L$ if $X \notin D_L$

We know that if $\text{append}(X, Y, T)$ is satisfiable, then $T \in D_T$. But, since $X \notin D_L$, i.e., $L(X)$ is false, $L(T)$ is false, i.e., $T \notin D_L$.

Requirements (5) and (6) are module dependent. For each prototype, we show it meets the two requirements separately.

For trace expressions

(a) In our prototypes, $I[e] = [] \in D_e$.

(b) Since any trace variable ϕ is interpreted as a Prolog list in our prototypes, $I[\phi] \in D_T$.

(c) For each pi in a procedure call $C(p1, p2, \dots, pn)$, we have a Prolog procedure $[C(p1, p2, \dots, pn)]$.

(d) It is clear that if $\text{append}(T, R, T1)$ is satisfiable, then $T1 = T.R$.

Thus, our prototypes meet requirement (7).

In above, we have illustrated that our Prolog prototypes meet the requirements on a trace model.

5.2.3. The Correctness of a Model

To prove that a model is a correct model for a trace specification, we have to show that all the assertions in the specification are true in the model [4]. In the trace methodology, specifications are expressed in terms of the legality, equivalence, and value of a single “new” call C following a normal form “history” T . For the legality assertion $L(T.C)$, in our model, we have

$$l(T.C) \text{ :- } nfdriver([], T.C, Nftr).$$

However, by the following *Idempotency Lemma* and its corollaries, we can prove that

$$\text{sat}(nfdriver([], T.C, Nftr)) \text{ if and only if } \text{sat}(nfdriver(T, C, Nftr)),$$

i.e., $nfdriver$ transforms a normal form trace T to itself. From Figure 16, we have

$$\begin{aligned} & nfdriver(Nftr, [], Nftr). \\ & nfdriver(Nf, [C|T1], Nftr) \text{ :-} \\ & \quad lp(Nf, [C]), \text{ nf}(Nf, [C], Nf1), \text{ nfdriver}(Nf1, T1, Nftr). \end{aligned}$$

Instead of discussing the satisfiability of $l(T)$, we can now discuss a much simpler problem: the satisfiability of lp and nf . Similarly, the *Idempotency Lemma* can simplify the proof that a value assertion $V(T.C)$ is true in a model.

Idempotency Lemma: Suppose that

- (1) $Nf.T$ is a normal form trace and U is any trace and
 - (2) for each single call C in T , $\text{sat}(lp(Nf, [C]))$ and $\text{sat}(nf(Nf, [C], Nftr))$,
- then
- $$\text{sat}(nfdriver(Nf, T.U, Nftr)) \text{ if and only if } \text{sat}(nfdriver(Nf.T, U, Nftr)).$$

Proof: Our proof is by induction on the length of T . For the basis step, when T 's length is 0 ($T = e$), it is clear that the following three procedures:

$\text{nfdriver}(Nf, T.U, Nftr),$
 $\text{nfdriver}(Nf, U, Nftr),$ and
 $\text{nfdriver}(Nf.T, U, Nftr)$

are either all satisfiable or all unsatisfiable.

For the induction hypothesis, suppose that for any normal form trace $Nf.T$ such that T is of length less than n , $\text{sat}(\text{nfdriver}(Nf, T.U, Nftr))$ if and only if $\text{sat}(\text{nfdriver}(Nf.T, U, Nftr))$. If T is of length n , we have

$\text{nfdriver}(Nf, T.U, Nftr) :-$
 $\text{lp}(Nf, [C1]), \text{nf}(Nf, [C1], Nf1), \text{nfdriver}(Nf1, T1.U, Nftr)$

where $T = C1.T1$.

Since $\text{sat}(\text{lp}(Nf, [C1]))$ and $\text{sat}(\text{nf}(Nf, [C1], Nf1))$. We have

$\text{sat}(\text{nfdriver}(Nf, T.U, Nftr))$ if and only if $\text{sat}(\text{nfdriver}(Nf1, T1.U, Nftr))$.

By the induction hypothesis, we have

$\text{sat}(\text{nfdriver}(Nf1, T1.U, Nftr))$ if and only if $\text{sat}(\text{nfdriver}(Nf1.T1, U, Nftr))$.

But, $Nf1.T1 = Nf.T$. Thus,

$\text{sat}(\text{nfdriver}(Nf, T.U, Nftr))$ if and only if $\text{sat}(\text{nfdriver}(Nf.T, U, Nftr))$. $\square$

Corollary 1: If T is a normal form trace and C is any call, then $\text{sat}(\text{nfdriver}([], T.C, Nftr))$ if and only if $\text{sat}(\text{nfdriver}(T, [C], Nftr))$.

Proof: This is a special case of the *Idempotency Lemma*, i.e., $Nf = e$ and $U = C$. $\square$

Corollary 2: For a normal form trace T and a call C , (1) if $\text{sat}(\text{lp}(T.C))$ then $\text{sat}(\text{lp}(T, [C]))$, and (2) if $\text{sat}(\text{v}(T.C, i))$ then $\text{sat}(\text{val}(T, C, i))$.

Proof: (1) $\text{sat}(\text{l}(T.C))$ if and only if $\text{sat}(\text{nfdriver}([], T.C, \text{Nftr}))$. From *Corollary 1*, we know that $\text{sat}(\text{nfdriver}([], T.C, \text{Nftr}))$ if and only if $\text{sat}(\text{nfdriver}(T, [C], \text{Nftr}))$. Since we have

```
nfdriver(T, [C], Nftr) :-
    lp(T, [C]), nf(T, [C], Nf1), nfdriver(Nf1, [], Nftr).
```

it is clear if $\text{sat}(\text{l}(T.C))$ then $\text{sat}(\text{lp}(T, [C]))$.

(2) Following the very same argument as in (1), $\text{sat}(\text{v}(T.C, i))$ if and only if $\text{sat}(\text{val}(T, C, i))$. $\square$

For a Prolog prototype, we rely on the above lemmas and corollaries to prove that all the assertions in its corresponding trace specification are true in the prototype.

5.3. Stack Prototype Correctness Proof

In this section, we give the *stack* example to illustrate how to apply the proof scheme to a specific prototype. We prove that the Prolog prototype shown in Figure 17, together with the procedures shown in Figure 16, is a correct model of the stack specification shown in Figure 5.

Step 1 We need to prove that *lp* and *nf* terminate in order to prove that *l* terminates. We assume that *append* terminates if any one of its parameters is instantiated. Then, from Figure 17, it is clear that *lp* and *nf* terminate.

Step 2 Since *append* terminates if any one of its parameter is instantiated, it is clear that *val* terminates. From *Corollary of the Termination Lemma*, we know $\text{nfdriver}([], T1, \text{Nftr})$ terminates ($T = T1.C$). Thus, *v* terminates. Now we prove that *val* is functional. Suppose that $\text{sat}(\text{val}(T, [\text{top}], V1))$ and $\text{sat}(\text{val}(T, [\text{top}], V2))$. Since we have

$\text{val}(T, [\text{top}], V) :- \text{append}(_, [\text{push}(V)], T),$

V is the parameter of the last *push* in T . Thus, both $V1$ and $V2$ are the parameter of the last *push* in T , i.e., $V1 = V2$; in another words, val is functional. Therefore, v is functional.

Step 3 We have two functions *length* and *count* in the specification. They both are functions from traces to integers. We use the *count* and Prolog built-in procedure *length* to define $I[\text{count}]$ and $I[\text{length}]$ respectively. They are functions from D_T to D_{INT} .

Step 4 For each single call C in a normalform trace T , we need to prove that $\text{sat}(\text{lp}(\text{Nf}, [C]))$ and $\text{sat}(\text{nf}(\text{Nf}, [C], \text{Nf1}))$ in order to show that the *Idempotency Lemma* applies. Since T is a normal form trace, C is a *push* call. Thus, from Figure 17, it is clear that $\text{sat}(\text{lp}(\text{Nf}, [\text{push}(_)]))$ and $\text{sat}(\text{nf}(\text{Nf}, [\text{push}(_)], \text{Nf1}))$.

Step 5 Assuming that T is a normal form trace and C a single call, we now prove that the seven assertions in the stack specification (See Figure 5) are true in our model.

1. $(\forall T)(\text{normalform}(T) \leftrightarrow \text{length}(T) = \text{count}(\text{push}, T))$

It is clear that if $\text{sat}(\text{normalform}(T))$, then T contains solely *push* calls. Thus, the number of *push* calls in T is the length of T .

2. $(\forall i)(C = \text{push}(i) \rightarrow L(T.C))$

Since we have

$\text{l}(T.C) :- \text{nfdriver}([], T.C, \text{Nftr})$

by *Corollary 1* of the *Idempotency Lemma*,

$\text{sat}(\text{l}(T.C))$ if and only if $\text{sat}(\text{nfdriver}(T, [C], \text{Nftr}))$

From Figure 16, we have

$\text{nfdriver}(T, [C], \text{Nftr}) :-$

$lp(T, [C]), nf(T, [C], Nftr),$
 $nfdriver(Nftr, [], Nftr).$

Since $nfdriver(Nftr, [], Nftr)$ is a fact, we now have

$sat(l(T.C))$ if and only if $sat(lp(T, [C]))$ and $sat(nf(T, [C], Nftr))$

From *Step 4*, we have both $sat(nf(T, [push(_)], Nftr))$ and
 $sat(lp(T, [push(_)]))$. Thus, $sat(l(T.C))$.

3. $C = pop \rightarrow (L(T.C) \leftrightarrow T \neq e)$

Since $L(T.C)$, $sat(l(T.C))$. By *Corollary 2* of the *Idempotency Lemma*, we have
 $sat(lp(T, C))$. $sat(lp(T, [pop]))$ if and only if T contains at least one *push* call,
that is, $T \neq e$.

4. $C = pop \rightarrow (L(T.C) \rightarrow (\forall T1, i)(T = T1.push(i) \rightarrow T.C \equiv T1))$

By the definition of equivalence, we have to prove that (i) for any trace U , $L(T.C.U)$ if
and only if $L(T1.U)$, and (ii) if $L(T.C.U)$ and U ends in a *top* call, then $V(T.C.U) =$
 $V(T1.U)$.

(i) For any U , we have

$$l(T.C.U) :- nfdriver([], T.C.U, Nftr) \quad (1)$$

By the *Idempotency Lemma*,

$$sat(nfdriver([], T.C.U, Nftr)) \text{ if and only if } sat(nfdriver(T, C.U, Nftr)) \quad (2)$$

Since we have

$$nfdriver(T, C.U, Nftr) :- lp(T, [C]), nf(T, C, Nf1), nfdriver(Nf1, U, Nftr)$$

and $L(T.C)$, $sat(lp(T, [C]))$, $sat(nf(T, C, Nf1))$, and $Nf1 = T1$. Thus,

$$\text{sat}(\text{nfdriver}(T, C.U, \text{Nftr})) \text{ if and only if } \text{sat}(\text{nfdriver}(T1, U, \text{Nftr})) \quad (3)$$

However, by the *Idempotency Lemma*,

$$\text{sat}(\text{nfdriver}(T1, U, \text{Nftr})) \text{ if and only if } \text{sat}(\text{nfdriver}([], T1.U, \text{Nftr})) \quad (4)$$

From Figure 16, we have

$$\text{sat}(l(T1.U) \text{ if and only if } \text{nfdriver}([], T1.U, \text{Nftr})) \quad (5)$$

Then, from (1) - (5), we have

$$\text{sat}(l(T.C.U)) \text{ if and only if } \text{sat}(l(T1.U))$$

(ii) Suppose that U is a trace ending in a *top* call, such that $U = U1.top$. Compare

$$\begin{aligned} v(T.C.U, V1) &:- \text{append}(T.C.U1, [\text{top}], T.C.U), \\ &\quad \text{nfdriver}([], T.C.U1, \text{Nftr1}), \\ &\quad \text{val}(\text{Nftr1}, [\text{top}], V1). \end{aligned}$$

and

$$\begin{aligned} v(T1.U, V2) &:- \text{append}(T1.U1, [\text{top}], T1.U), \\ &\quad \text{nfdriver}([], T1.U1, \text{Nftr2}), \\ &\quad \text{val}(\text{Nftr2}, [\text{top}], V2). \end{aligned}$$

It is clear that the two *append* sub-goals are satisfiable. Therefore, if $\text{Nftr1} = \text{Nftr2}$, then $V1$ and $V2$ are instantiated as the same value. By the *Idempotency Lemma*, we have (a) $\text{sat}(\text{nfdriver}([], T.C.U1, \text{Nftr1}))$ if and only if $\text{sat}(\text{nfdriver}(T, C.U1, \text{Nftr1}))$, and (b) $\text{sat}(\text{nfdriver}([], T1.U1, \text{Nftr2}))$ if and only if $\text{sat}(\text{nfdriver}(T1, U1, \text{Nftr2}))$. From (3) in the proof of (i), $\text{sat}(\text{nfdriver}(T, C.U1, \text{Nftr1}))$ if and only if $\text{sat}(\text{nfdriver}(T1, U1, \text{Nftr1}))$.

Then, $\text{Nftr1} = \text{Nftr2}$ and hence $V1 = V2$.

$$5. C = \text{top} \rightarrow (L(T.C) \leftrightarrow T \neq e)$$

Since $L(T.C)$, $\text{sat}(lp(T.C))$. By *Corollary 2* of the *Idempotency Lemma*, $\text{sat}(lp(T.C))$.

$\text{sat}(lp(T, [\text{pop}]))$ if and only if T contains at least one push call, that is, $T \neq e$.

6. $C = \text{top} \rightarrow (L(T.C) \rightarrow T.C \equiv T)$

By the definition of equivalence, we have to prove that (i) $L(T.C)$ if and only if $L(T)$, and (ii) for any trace U that ends in a *top* call and both $L(T.C.U)$ and $L(T.U)$, $V(T.C.U) = V(T.U)$.

(i) For any U , we have

$$lp(T.C.U) :- \text{nfdriver}([], T.C.U, \text{Nftr}) \quad (1)$$

By the *Idempotency Lemma*,

$$\text{sat}(\text{nfdriver}([], T.C.U, \text{Nftr})) \text{ if and only if } \text{sat}(\text{nfdriver}(T, C.U, \text{Nftr})) \quad (2)$$

Since we have

$$\begin{aligned} \text{nfdriver}(T, C.U, \text{Nftr}) :- \\ lp(T, [C]), \text{nf}(T, C, \text{Nf1}), \text{nfdriver}(\text{Nf1}, U, \text{Nftr}) \end{aligned}$$

and $L(T.C)$, $\text{sat}(lp(T, [C]))$, $\text{sat}(\text{nf}(T, C, \text{Nf1}))$, and $\text{Nf1} = T$. Thus,

$$\text{sat}(\text{nfdriver}(T, C.U, \text{Nftr})) \text{ if and only if } \text{sat}(\text{nfdriver}(T, U, \text{Nftr})) \quad (3)$$

However, by the *Idempotency Lemma*,

$$\text{sat}(\text{nfdriver}(T, U, \text{Nftr})) \text{ if and only if } \text{sat}(\text{nfdriver}([], T.U, \text{Nftr})) \quad (4)$$

From Figure 16, we have

$$\text{sat}(lp(T.U)) \text{ if and only if } \text{nfdriver}([], T.U, \text{Nftr}) \quad (5)$$

Then, from (1) - (5), we have

$\text{sat}(l(T.C.U))$ if and only if $\text{sat}(l(T.U))$

(ii) Suppose that U is a trace ending in *top* call, such that $U = U1.top$. Compare

```
v(T.C.U, V1) :- append(T.C.U1, [top], T.C.U),
                nfdriver([], T.C.U1, Nftr1),
                val(Nftr1, [top], V1).
```

and

```
v(T.U, V2) :- append(T.U1, [top], T1.U),
               nfdriver([], T.U1, Nftr2),
               val(Nftr2, [top], V2).
```

It is clear that the two `append` sub-goals are satisfiable. Therefore, if $Nftr1 = Nftr2$, then $V1$ and $V2$ are instantiated as the same value. By the *Idempotency Lemma*, we have (a) $\text{sat}(\text{nfdriver}([], T.C.U1, Nftr1))$ if and only if $\text{sat}(\text{nfdriver}(T, C.U1, Nftr1))$, and (b) $\text{sat}(\text{nfdriver}([], T.U1, Nftr2))$ if and only if $\text{sat}(\text{nfdriver}(T, U1, Nftr2))$. From (3) in the proof of (i), $\text{sat}(\text{nfdriver}(T, C.U1, Nftr1))$ if and only if $\text{sat}(\text{nfdriver}(T, U1, Nftr1))$. Then, $Nftr1 = Nftr2$ and hence $V1 = V2$.

7. $C = \text{top} \rightarrow (L(T.C) \rightarrow (\forall T1, i)(T = T1.\text{push}(i) \rightarrow V(T.C) = i))$

Since $L(T.C)$, $\text{sat}(l(T, C))$. By *Corollary 2* of the *Idempotency Lemma*, $\text{sat}(lp(T, C))$. $\text{sat}(lp(T, C))$ if and only if T contains at least one *push* calls. Thus, $\text{sat}(\text{append}(_, [\text{push}(i)], T))$. Since we have

```
val(T, [top], V) :- append(\_, [push(V)], T).
```

V is instantiated as i . From Figure 16, we have

```
v(T.C, V) :-
    append(T, C, T.C),
    nfdriver([], T.C, Nftr),
    val(Nftr, C, V).
```

Thus, $\text{sat}(\nu(T.C, i))$.

From Step 1 to Step 5, we have proved that the stack prototype is a correct model of the stack trace specification.

CHAPTER 6

Deterministic And Non-Deterministic Specifications

An inconsistent specification fails by placing too many constraints on its intended implementation. A specification can also fail by placing too few constraints, leaving the implementor in the air about module behavior. However, for some modules, some freedom in return values is appropriate. For example, in the *priority queue* module, if there are two priority-data pairs with the same priority, we do not care which one is returned by *front*. We say a trace specification is *deterministic* (in [4], referred to as *total*) if it asserts explicitly what output should be generated by each legal trace ending in a function call. In Section 6.1, we present a scheme for proving that a specification is deterministic, illustrated on the *stack* specification (see Figure 5). A scheme used to prove that a specification is non-deterministic is presented in Section 6.2, and demonstrated on the *priority queue* specification.

6.1. Deterministic Specifications

To prove that a specification is deterministic, we must show that, for any legal trace T ending in a function call, there is a derivation of $V(T) = a$ (a is a constant) from the specification [4]. McLean proves a specification deterministic by induction on trace length [4]. In the trace methodology, every trace is associated with its normal form equivalent. When we discuss $V(T)$, we discuss the return value of T 's equivalent normal form trace. If every value assertion in a specification is deterministic, then the full specification is deterministic. Thus, to prove that a specification is deterministic, we examine every function call C in the specification and try to

derive $V(T.C) = a$, assuming that $L(T.C)$ and that T is a normal form trace.

Now we prove that the *stack* specification shown in Figure 5 is a deterministic specification. Suppose that T is a normal form trace. It is clear that if T is of length greater than 0, then $V(T.top)$ always returns the integer i that is passed by the rightmost *push* call in T . Since the rightmost *push* call is unique in T , we can derive only one value (from the last assertion of the specification) for any legal trace ending in a *top* call. Since *top* is the only function call in the specification, the specification is deterministic.

6.2. Non-Deterministic Specifications

Assuming that $L(T.C)$, that C is a function call, and that T is a normal form trace, we prove that a specification is non-deterministic, by building two models that disagree on what to assign $V(T.C)$. In this section, we will use this method to prove that the *prqueue* specification is non-deterministic.

Using the same technique described in Chapter 4, we build two Prolog models for the *prqueue* module: PQ1 and PQ2. Except for the *parse* procedures, the two models are the same. Their common procedures are shown in Figure 25.

The *parse* procedure takes four parameters. The first parameter P is the priority for the newly inserted data. The second one, T , is a list corresponding to a normal form trace. *parse* is satisfiable if and only if (1) the variable $T1$ is instantiated such that for all $insert(X, Y)$ in $T1$, $P \geq X$, (2) the variable $T2$ is instantiated such that for all $insert(X, Y)$ in $T2$, $P \leq X$, and (3) $append(T1, T2, T)$ is satisfiable. The *parse* procedures shown in Figure 26 and Figure 27 are used in PQ1 and PQ2 respectively.

Figure 25. Common Procedures in PQ1 and PQ2

```

normalform([]).
normalform([insert(_,_)]).
normalform([insert(P,_)|T]) :-
    append([insert(P1,_)|_,T], P =< P1, normalform(T).

lp(T, [insert(_,_)]).
lp(T, [remove]) :- append([insert(_,_)|_,T].
lp(T, [front]) :- append(_, [insert(_,_)|T].

nf(T, [insert(P,I)|Nftr] :-
    parse(P,T,T1,T2), append(T1, [insert(P,I)|Temp],
    append(Temp,T2,Nftr).
nf(T, [remove]|Nftr) :- append(Nftr, [insert(_,_)|T].
nf(T, [front]|Nftr) :- append([],T,Nftr).

val(T, [front],V) :- append(_, [insert(_,V)|T].

```

For the parse procedure used in PQ1, T1 is instantiated such that for all insert(X,Y) in T1, $P > X$ and T2 is instantiated such that for all insert(X,Y) in T2, $P \leq X$. If we have

T = [insert(1,2),insert(2,3),insert(2,4)] and P = 2

then, parse(P,T,T1,T2) is satisfiable if and only if

T1 = [insert(1,2)] and T2 = [insert(2,3),insert(2,4)]

Figure 26. Parse Procedures in PQ1

```

parse(P,[],[],[]).
parse(P,T,T,[]) :- append(_, [insert(X,_)|T], X < P.
parse(P,T,T1,T2) :-
    append(T11, [insert(X,Y)|T], X >= P,
    parse(P,T11,T1,T22), append(T22, [insert(X,Y)|T2]).

```

Figure 27. Parse Procedures in PQ2

```

parse(P, [], [], []).
parse(P, T, T, []) :- append(_, [insert(X, _)], T), X =< P.
parse(P, T, T1, T2) :-
    append(T11, [insert(X, Y)], T), X > P,
    parse(P, T11, T1, T22), append(T22, [insert(X, Y)], T2) .

```

In PQ1, $V(\text{insert}(1,2).\text{insert}(1,3).\text{front}) = 2$.

For the parse procedures used in PQ2, T1 is instantiated such that for all $\text{insert}(X, Y)$ in T1, $P \geq X$ and T2 is instantiated such that for all $\text{insert}(X, Y)$ in T2, $P < X$. If we have

$T = [\text{insert}(1, 2), \text{insert}(2, 3), \text{insert}(2, 4)]$ and $P = 2$

then, $\text{parse}(P, T, T1, T2)$ is satisfiable if and only if

$T1 = [\text{insert}(1, 2), \text{insert}(2, 3), \text{insert}(2, 4)]$ and $T2 = []$

In PQ2, $V(\text{insert}(1,2).\text{insert}(1,3).\text{front}) = 3$.

Since the two models have different behaviors on return value of a trace, the specification is non-deterministic.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

From our specification experience, we feel that the trace methodology provides us with a systematic way to write trace specification for modules. According to the principle of “separation of concerns”, in each step in writing a specification, we only consider a single procedure call. This helps us to avoid both redundancy and inconsistency to a certain degree. The correctness of a specification is much easier to achieve than with ad hoc trace specification.

Our model building scheme also allows us to develop executable Prolog models quickly and reliably. The scheme is simple, and uniform across different specifications written in the trace methodology. The correspondence between predicate logic assertions and Prolog procedures is clear and intuitive. The model interfaces we developed facilitate a considerable freedom in exercising the Prolog models.

Our model correctness proof scheme provides a systematic way to organize the correctness proof. It is directly based on the model building scheme, and hence is also simple and uniform across different models. The *Termination Lemma*, *Idempotency Lemma* and their corollaries help us reduce the complexity of a proof.

We have also presented a scheme for showing that a specification is deterministic or non-deterministic. Based on the facilities provided by the trace methodology, the scheme is simple and straightforward.

The weakness of our method is that we do not have a precise and formal translation scheme: all the translations rely on our intuitive understanding of trace assertions. Since the

existence of a Prolog prototype can not guarantee the consistency of its specification, our prototype building scheme alone is not sufficient to provide the consistency proofs for specifications. What is missing here is a precisely developed translation scheme, which can prevent two contradictory assertions in a specification from being translated into a Prolog prototype. Nevertheless, our scheme is a step in that direction and it does provides us with a systematic way to build Prolog prototypes.

We see two areas for future research. We need a formal and precise translation scheme from predicate logic assertions to their corresponding Prolog procedures. Together with our model building scheme, they provide a precise way to prove the consistency of specifications. Automatic generation of Prolog prototypes directly from specifications would significantly improve the practical utility of the technique.

BIBLIOGRAPHY

- [1] Hoffman, D.M., Snodgrass, R.T., Trace Specifications: Methodology and Models (accepted for publication) *IEEE Transactions on Software Engineering*.
- [2] Parnas, D.L., Software Engineering Principles. Lecture notes. University of Victoria, Department of Computer Science, (Feb. 1983).
- [3] Bartussek, W., Parnas, D.L., Using Assertions About Traces to Write Abstract Specifications for Software Modules. In *Second Conference of the European Cooperation in Informatics*, Springer-Verlag (1978).
- [4] McLean, J.A., A Formal Method for the Abstract Specification of Software. *J. ACM* 31, 3, (July 1984).
- [5] Goguen, J., Meseguer, J., Rapid Prototyping in the OBJ Executable Specification Language. *Software Engineering Notes* 7, 5 (Dec. 1982).
- [6] Clocksin, W.F., Mellish, C.S., *Programming in Prolog*. Springer-Verlag, (1981).
- [7] Ambler, A.L., Good, D.I., Browne, J.C., Burger, W.F., Cohen, R.M., Hoch, C.G., Wells, R.E., GYPSY: A Language for Specification and Implementation of Verifiable Programs. *Proceedings of the ACM Conference on Language Design for Reliable Software, SIG-PLAN Notices*, 12 (3). (1977).

- [8] Dixon, J., McLean, J., Parnas, D., Rapid Prototyping by Means of Abstract Module Specifications Written as Trace Axioms. *Software Engineering Notes* 7, 5 (Dec. 1982).
- [9] Parnas, D.L., A Technique for Software Specification with Modules. *CACM*, 15 (5) (May, 1972).
- [10] Guttag, J.V., Notes on Type Abstraction. *Proceedings of Conference on Specifications of Reliable Software*. (1979).
- [11] Guttag, J.V., Horinig, J.J., The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10 (1) (1978).
- [12] Moitra, A., Direct Implementation of Algebraic Specification of Abstract Data Types. *IEEE Transactions on Software Engineering*, se-8,1. (Jan. 1982).
- [13] Hoffman, D.M., Practical Interface Specification. Technical Report, Department of Computer Science, University of Victoria, (Dec. 1987).
- [14] Hoffman, D.M., Trace Specification of Communications Protocols. Ph.D. Dissertation, University of North Carolina, Computer Science Department, (1984).
- [15] Kernighan, B.W., Ritchie, D.M., *The C Programming Language*. Prentice-Hall, Inc. (1978).
- [16] Horspool, R.N., *C Programming in the Berkeley Unix Environment*. Prentice-Hall Canada (1986).

APPENDIX A

TEN TRACE SPECIFICATIONS

1. Stack

NAME

stack

SYNTAX

push: integer;
pop: ;
top: → integer;

PREDICATES

normalform(T) $\leftrightarrow$ length(T) = count(push,T)

SEMANTICS

$(\forall T, C)(\text{normalform}(T) \rightarrow$
 $(\forall i)(C = \text{push}(i) \rightarrow$
 /*L*/ L(T.C)
) &
 $(C = \text{pop} \rightarrow$
 /*L*/ $(L(T.C) \leftrightarrow T \neq e)$ &
 /*E*/ $(L(T.C) \rightarrow$
 $(\forall T1, i)(T = T1.\text{push}(i) \rightarrow T.C \equiv T1))$
) &
 $(C = \text{top} \rightarrow$
 /*L*/ $(L(T.C) \leftrightarrow T \neq e)$ &
 $(L(T.C) \rightarrow$
 /*E*/ $T.C \equiv T$ &
 /*V*/ $(\forall T1, i)(T = T1.\text{push}(i) \rightarrow V(T.C) = i))$)

2. Queue

NAME

queue

SYNTAX

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

PREDICATES

normalform(T) $\leftrightarrow$ length(T) = count(add,T)

SEMANTICS

$(\forall T, C)(\text{normalform}(T) \rightarrow$
 $(\forall i)(C = \text{add}(i) \rightarrow$
 $\text{/*L*/ } L(T.C)$
 $) \&$
 $(C = \text{remove} \rightarrow$
 $\text{/*L*/ } (L(T.C) \leftrightarrow T \neq e) \&$
 $\text{/*E*/ } (L(T.C) \rightarrow$
 $(\forall T1, i)(T = \text{add}(i).T1 \rightarrow T.C \equiv T1))$
 $) \&$
 $(C = \text{front} \rightarrow$
 $\text{/*L*/ } (L(T.C) \leftrightarrow T \neq e) \&$
 $(L(T.C) \rightarrow$
 $\text{/*E*/ } T.C \equiv T \&$
 $\text{/*V*/ } (\forall T1, i)(T = \text{add}(i).T1 \rightarrow V(T.C) = i))))$

3. Overflow Stack

NAME

ostack

SYNTAX

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

PREDICATES

normalform(T) $\leftrightarrow$ length(T) = count(push,T) & length(T) $\leq$ STACKSIZE

SEMANTICS

$(\forall T, C)(\text{normalform}(T) \rightarrow$
 $(\forall i)(C = \text{push}(i) \rightarrow$
 $\text{/*L*/ } L(T.C) \&$
 $(L(T.C) \rightarrow$
 $\text{/*E*/ } (\text{length}(T.C) > \text{STACKSIZE} \rightarrow$

$$\begin{aligned}
& (\forall T1, i)(T = \text{push}(i).T1 \rightarrow T.C \equiv T1.C)) \\
&) \& \\
& (C = \text{pop} \rightarrow \\
& /*L*/ \quad (L(T.C) \leftrightarrow T \neq e) \& \\
& \quad (L(T.C) \rightarrow \\
& /*E*/ \quad (\forall T1, i)(T = T1.\text{push}(i) \rightarrow T.C \equiv T1.C))) \\
&) \& \\
& (C = \text{top} \rightarrow \\
& /*L*/ \quad (L(T.C) \leftrightarrow T \neq e) \& \\
& \quad (L(T.C) \rightarrow \\
& /*E*/ \quad (T.C \equiv T) \& \\
& /*V*/ \quad (\forall T1, i)(T = T1.\text{push}(i) \rightarrow V(T.C) = i)))
\end{aligned}$$

4. Sorting Queue

NAME

sorting_queue

SYNTAX

add: integer;

remove: ;

front: $\rightarrow$ integer;

PREDICATES

normalform(T) $\leftrightarrow$ length(T) = count(add, T) &
 $(\forall T1, T2, T3, i, j)(T = T1.\text{add}(i).T3.\text{add}(j).T2 \rightarrow i \leq j)$

SEMANTICS

$$\begin{aligned}
& (\forall T, C)(\text{normalform}(T) \rightarrow \\
& \quad (\forall i)(C = \text{add}(i) \rightarrow \\
& \quad /*L*/ \quad L(T.C) \& \\
& \quad \quad (L(T.C) \rightarrow \\
& \quad /*E*/ \quad (\forall T1, T2)(T = T1.T2 \& \text{normalform}(T1.C.T2) \rightarrow \\
& \quad \quad \quad T.C \equiv T1.C.T2)) \\
& \quad) \& \\
& \quad (C = \text{remove} \rightarrow \\
& \quad /*L*/ \quad (L(T.C) \leftrightarrow T \neq e) \& \\
& \quad \quad (L(T.C) \rightarrow \\
& \quad /*E*/ \quad (\forall T1, i)(T = T1.\text{add}(i)) \rightarrow T.C \equiv T1) \\
& \quad) \& \\
& \quad (C = \text{front} \rightarrow \\
& \quad /*L*/ \quad (L(T.C) \leftrightarrow T \neq e) \& \\
& \quad \quad (L(T.C) \rightarrow (\\
& \quad /*E*/ \quad T.C \equiv T \&
\end{aligned}$$

*/*V*/* $(\forall T1,i)(T = T1.add(i) \rightarrow V(T.C) = i))$

5. Table List

NAME

tblist

SYNTAX

insert: integer;
 append: integer;
 delete: ;
 goleft: ;
 goright: ;
 current: $\rightarrow$ integer;

PREDICATES

normalform(T) $\leftrightarrow$
 $(\exists T1,T2)(\text{prefix}(T,T1.T2) \ \& \ \text{all}(\text{append},T1) \ \& \ \text{all}(\text{goleft},T2) \ \& \ \text{length}(T1) > \text{length}(T2))$

parse(T1,C,T2,G) $\leftrightarrow$
 $\text{all}(\text{append},T1.C.T2) \ \& \ \text{all}(\text{goleft},G) \ \& \ \text{length}(C) = 1 \ \& \ \text{length}(T2) = \text{length}(G)$

$\text{all}(C,T) \leftrightarrow \text{count}(C,T) = \text{length}(T)$

SEMANTICS

$(\forall T,C)(\text{normalform}(T) \rightarrow$
 $(\forall T1,C1,T2,G)(T = T1.C1.T2.G \ \& \ \text{parse}(T1,C1,T2,G) \rightarrow$
 $(\forall i)(C = \text{append}(i) \rightarrow$
 $\textit{/*L*/} \quad L(T.C) \ \& \$
 $(L(T.C) \rightarrow$
 $\textit{/*E*/} \quad T.C \equiv T1.C1.C.T2.G)$
 $) \ \& \$
 $(\forall i)(C = \text{insert}(i) \rightarrow$
 $\textit{/*L*/} \quad L(T.C) \ \& \$
 $\textit{/*E*/} \quad (L(T.C) \rightarrow$
 $(T = e \rightarrow T.C \equiv \text{append}(i)) \ \& \$
 $(T \neq e \rightarrow T.C \equiv T1.append(i).C1.T2.G.goleft))$
 $) \ \& \$
 $(C = \text{delete} \rightarrow$
 $\textit{/*L*/} \quad (L(T.C) \leftrightarrow C \neq e) \ \& \$
 $\textit{/*E*/} \quad (L(T.C) \rightarrow$
 $(G = e \rightarrow T.C \equiv T1.T2) \ \& \$

```

        (G = G1.goleft → T.C ≡ T1.T2.G1))
    ) &
    (C = goleft →
/*L*/    (L(T.C) ↔ normalform(T.C))
    ) &
    (C = goright →
/*L*/    (L(T.C) ↔ G ≠ e) &
        (L(T.C) →
/*E*/    (∀ G1)(G = G1.goleft → T.C ≡ T1.C1.T2.G1))
    ) &
    (C = current →
/*L*/    (L(T.C) ↔ G ≠ e) &
        (L(T.C) →
/*E*/    T.C ≡ T &
            (∀ i)(C1 = append(i) → V(T.C) = i)))
))

```

6. Unique

NAME

unique

SYNTAX

getint: → integer;

PREDICATES

normalform(T) ↔ true

SEMANTICS

```

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

```

7. Traversing Stack

NAME

travstack

SYNTAX

push: integer;

pop: ;

down: ;
 current: $\rightarrow$ integer;
 to_top: ;

PREDICATES

$\text{normalform}(T) \leftrightarrow (\exists T1, T2)(\text{prefix}(T, T1.T2) \ \& \ \text{all}(\text{push}, T1) \ \& \ \text{all}(\text{down}, T2) \ \& \ \text{length}(T1) > \text{length}(T2))$

$\text{parse}(T1, C, T2, D) \leftrightarrow (T1.C.T2.D = e \mid \text{all}(\text{push}, T1.C.T2) \ \& \ \text{all}(\text{down}, D) \ \& \ \text{length}(C) \leq 1 \ \& \ \text{length}(T2) = \text{length}(D))$

$\text{all}(C, T) \leftrightarrow \text{count}(C, T) = \text{length}(T)$

SEMANTICS

$(\forall T, C)(\text{normalform}(T) \rightarrow$
 $(\forall T1, C1, T2, D)(T = T1.C1.T2.D \ \& \ \text{parse}(T1, C1, T2, D) \rightarrow$
 $(\forall i)(C = \text{push}(i) \rightarrow$
 $\text{/*L*/} \quad (L(T.C) \leftrightarrow D = e)$
 $) \ \&$
 $(C = \text{pop} \rightarrow$
 $\text{/*L*/} \quad (L(T.C) \leftrightarrow D = e \ \& \ C1 \neq e) \ \&$
 $\text{/*E*/} \quad (L(T.C) \rightarrow T.C \equiv T1)$
 $) \ \&$
 $(C = \text{down} \rightarrow$
 $\text{/*L*/} \quad (L(T.C) \leftrightarrow \text{normalform}(T.C))$
 $) \ \&$
 $(C = \text{current} \rightarrow$
 $\text{/*L*/} \quad (L(T.C) \leftrightarrow C1 \neq e) \ \&$
 $\quad L(T.C) \rightarrow ($
 $\text{/*E*/} \quad T.C \equiv T \ \&$
 $\text{/*V*/} \quad (\forall i)(C1 = \text{push}(i) \rightarrow V(T.C) = i))$
 $) \ \&$
 $(C = \text{to_top} \rightarrow$
 $\text{/*L*/} \quad L(T.C) \ \&$
 $\text{/*E*/} \quad (L(T.C) \rightarrow T.C \equiv T1.C1.T2))))))$

8. Priority Queue

NAME

priqueue

SYNTAX

insert: integer integer;

remove: ;
 front: → integer;

PREDICATES

normalform(T) $\leftrightarrow$ length(T) = count(insert,T) &
 $(\forall T1, T2, T3, p, d, p1, d1)$
 $(T = T1.insert(p, d).T2.insert(p1, d1).T3 \rightarrow p \leq p1)$

SEMANTICS

$(\forall T, C)(\text{normalform}(T) \rightarrow$
 $(\forall i)(C = \text{insert}(p, i) \rightarrow$
 $/*L*/ \quad L(T.C) \&$
 $(L(T.C) \rightarrow$
 $/*E*/ \quad (\forall T1, T2)(T = T1.T2 \& \text{normalform}(T1.C.T2) \rightarrow$
 $\quad T.C \equiv T1.C.T2))$
 $) \&$
 $(C = \text{remove} \rightarrow$
 $/*L*/ \quad (L(T.C) \leftrightarrow T \neq e) \&$
 $(L(T.C) \rightarrow$
 $/*E*/ \quad (\forall T1, p, d)(T = T1.insert(p, d) \rightarrow T.C \equiv T1))$
 $) \&$
 $(C = \text{front} \rightarrow$
 $/*L*/ \quad (L(T.C) \leftrightarrow T \neq e) \&$
 $(L(T.C) \rightarrow$
 $/*E*/ \quad (T.C \equiv T) \&$
 $/*V*/ \quad (\forall T1, p, d)(T = T1.insert(p, d) \rightarrow V(T.C) = d))))$

9. Shortest Path

NAME

spath

SYNTAX

init: integer;
 getmaxn: → integer;
 addnode: integer;
 exnode: integer → boolean;
 delnode: integer;
 addedge: integer integer integer;
 exedge: integer integer → boolean;
 deledge: integer integer;
 edgelen: integer integer → integer;
 expath: integer integer → boolean;
 pathlen: integer integer → integer;

firststop: integer integer $\rightarrow$ integer;

PREDICATES

$$\begin{aligned} \text{normalform}(T) \leftrightarrow & (\exists T1, T2, \text{maxn})(\text{prefix}(T, \text{init}(\text{maxn}).T1.T2) \ \& \\ & \text{all}(\text{addnode}, T1) \ \& \ \text{all}(\text{addege}, T2) \ \& \\ & (\forall T11, T12, n)(T1 = T11.\text{addnode}(n).T12 \rightarrow \\ & \quad n \leq \text{maxn}) \ \& \\ & \neg(\exists T11, T12, T13, n) \\ & \quad (T1 = T11.\text{addnode}(n).T12.\text{addnode}(n).T13) \ \& \\ & (\forall T21, T22, s, d, l)(T2 = T21.\text{addege}(s, d, l).T22 \rightarrow \\ & \quad \text{legalnode}(T1, s) \ \& \ \text{legalnode}(T1, d) \ \& \ s \neq d) \ \& \\ & \neg(\exists T21, T22, T23, s, d, l, l1) \\ & \quad (T2 = T21.\text{addege}(s, d, l).T22.\text{addege}(s, d, l1).T23)) \end{aligned}$$

$$\text{all}(C, T) \leftrightarrow \text{length}(T) = \text{count}(C, T)$$

$$\text{inrange}(T, n) \leftrightarrow (\exists \text{maxn}, T1)(T = \text{init}(\text{maxn}).T1 \ \& \ n \leq \text{maxn})$$

$$\text{legalnode}(T, n) \leftrightarrow (\exists T1, T2)(T = T1.\text{addnode}(n).T2)$$

$$\text{legaledge}(T, s, d) \leftrightarrow (\exists T1, T2, l)(T = T1.\text{addege}(s, d, l).T2)$$

SEMANTICS

$$\begin{aligned} (\forall T, C)(\text{normalform}(T) \ \& \ \text{length}(C) = 1 \rightarrow \\ & (\forall n, d)(C = \text{init}(n) \rightarrow \\ & \quad /*L*/ \quad (L(T.C) \leftrightarrow n > 0) \ \& \\ & \quad (L(T.C) \rightarrow \\ & \quad /*E*/ \quad T.C \equiv C) \\ & \quad) \ \& \\ & (C = \text{getmaxn} \rightarrow \\ & \quad /*L*/ \quad (L(T.C) \leftrightarrow T \neq e) \ \& \\ & \quad (L(T.C) \rightarrow \\ & \quad /*E*/ \quad (T.C \equiv T) \ \& \\ & \quad /*V*/ \quad (\forall T1, n)(T = \text{init}(n).T1 \rightarrow V(T.C) = \text{maxn})) \\ & \quad) \ \& \\ & (\forall n)(C = \text{addnode}(n) \rightarrow \\ & \quad /*L*/ \quad (L(T.C) \leftrightarrow \text{inrange}(T, n) \ \& \ \neg \text{legalnode}(T, n)) \ \& \\ & \quad (L(T.C) \rightarrow \\ & \quad (\forall T1, T2)(T = T1.T2 \ \& \ \text{normalform}(T1.C.T2) \rightarrow \\ & \quad /*E*/ \quad T.C \equiv T1.C.T2)) \\ & \quad) \ \& \\ & (\forall n)(C = \text{exnode}(n) \rightarrow \\ & \quad /*L*/ \quad (L(T.C) \leftrightarrow \text{inrange}(T, n)) \ \& \\ & \quad (L(T.C) \rightarrow \end{aligned}$$


```

/*E*/      (T.C ≡ T) &
/*V*/      (V(T.C) ↔ legalnode(T,n)))
) &
(∀ n)(C = delnode(n) →
/*L*/      (L(T.C) ↔ legalnode(T,n)) &
            (L(T.C) →
/*E*/      (∀ T1,T2,n)(T = T1.addnode(n).T2 → T.C ≡ T1.T2))
) &
(∀ s,d,l)(C = addedge(s,d,l) →
/*L*/      (L(T.C) ↔ normalform(T.C))
) &
(∀ s,d)(C = exedge(s,d) →
/*L*/      (L(T.C) ↔ inrange(T,s) & inrange(T,d)) &
            (L(T.C) →
/*E*/      (T.C ≡ T) &
/*V*/      (V(T.C) ↔ (legaledge(T,s,d) | s = d))
) &
(∀ s,d)(C = deledge(s,d) →
/*L*/      (L(T.C) ↔ legaledge(T,s,d)) &
            (L(T.C) →
/*E*/      (∀ T1,T2,l)(T = T1.addedge(s,d,l).T2 → T.C ≡ T1.T2))
) &
(∀ s,d)(C = edgelen(s,d) →
/*L*/      (L(T.C) ↔ (legaledge(T,s,d) | s = d)) &
            (L(T.C) →
/*E*/      (T.C ≡ T) &
/*V*/      (s = d → V(T.C) = 0) &
            (s ≠ d → (∀ T1,T2,l)
                    (T = T1.addedge(s,d,l).T2 → V(T.C) = 1)))
) &
(∀ s,d)(C = expath(s,d) →
/*L*/      (L(T.C) ↔ legalnode(s) & legalnode(d)) &
            (L(T.C) →
/*E*/      (T.C ≡ T) &
/*V*/      (V(T.C) ↔ (s = d |
                    (∃ T1,T2,n)(T = T1.addedge(s,n).T2 &
                    V(T.expath(n,d)))))
            )
) &
(∀ s,d)(C = pathlen(s,d) →
/*L*/      (L(T.C) ↔ V(T.expath(s,d))) &
            (L(T.C) →
/*E*/      (T.C ≡ T) &
/*V*/      (s = d → V(T.C) = 0) &
            (s ≠ d → (∀ T1,T2,n,l)

```

$$\begin{aligned}
 & (\exists Tl1)(Tl = mkl.Tl1.mvp \& \text{match}(Tl1, Tail, Tc)) \\
 &) \& \\
 & (C = mvr \rightarrow \\
 & \quad (\exists Tr1)(Tr = mkr.Tr1.mvp \& \text{match}(Tr1, Tail, Tc)) \\
 &) \\
 &))
 \end{aligned}$$

$$\begin{aligned}
 \text{tree}(T) & \leftrightarrow (\exists Trt, Tl, Tr) \\
 & (T = Trt.Tl.Tr \& \text{root}(Trt) \& \text{ltree}(Tl) \& \text{rtree}(Tr))
 \end{aligned}$$

$$\text{root}(T) \leftrightarrow (T = e \mid (\exists i)(T = \text{setc}(i)))$$

$$\text{ltree}(T) \leftrightarrow (T = e \mid (\exists T1)(T = mkl.T1.mvp \& \text{tree}(T1)))$$

$$\text{rtree}(T) \leftrightarrow (T = e \mid (\exists T1)(T = mkr.T1.mvp \& \text{tree}(T1)))$$

$$\text{allmv}(Tm) \leftrightarrow (\text{count}(mvl, Tm) + \text{count}(mvr, Tm) = \text{length}(Tm))$$

SEMANTICS

$$\begin{aligned}
 & (\forall T, C)(\text{normalform}(T) \rightarrow \\
 & (\forall T1, Tc, T2, Tm)(T = T1.Tc.T2.Tm \& \text{parse}(T1, Tc, T2, Tm) \rightarrow \\
 & (\forall Tcrt, Tcl, Tcr)(Tc = Tcrt.Tcl.Tcr \& \text{tree}(Tc) \& \\
 & \quad \text{root}(Tcrt) \& \text{ltree}(Tcl) \& \text{rtree}(Tcr) \rightarrow \\
 & \quad (\forall i)(C = \text{setc}(i) \rightarrow \\
 & \quad \quad /*L*/ \quad L(T.C) \& \\
 & \quad \quad /*E*/ \quad (L(T.C) \rightarrow T.C \equiv T1.C.Tcl.Tcr.T2.Tm) \\
 & \quad) \& \\
 & \quad (C = \text{getc} \rightarrow \\
 & \quad \quad /*L*/ \quad (L(T.C) \leftrightarrow Tcrt \neq e) \& \\
 & \quad \quad /*E*/ \quad (L(T.C) \rightarrow \\
 & \quad \quad \quad (T.C \equiv T) \& \\
 & \quad \quad /*V*/ \quad (\forall i)(Tcrt = \text{setc}(i) \rightarrow V(T.C) = i)) \\
 & \quad) \& \\
 & \quad (C = \text{exl} \rightarrow \\
 & \quad \quad /*L*/ \quad L(T.C) \& \\
 & \quad \quad /*E*/ \quad (L(T.C) \rightarrow \\
 & \quad \quad \quad T.C \equiv T \& \\
 & \quad \quad /*V*/ \quad (V(T.C) \leftrightarrow Tcl \neq e)) \\
 & \quad) \& \\
 & \quad (C = \text{exr} \rightarrow \\
 & \quad \quad /*L*/ \quad L(T.C) \& \\
 & \quad \quad /*E*/ \quad (L(T.C) \rightarrow \\
 & \quad \quad \quad T.C \equiv T \& \\
 & \quad \quad /*V*/ \quad (V(T.C) \leftrightarrow Tcr \neq e)) \\
 & \quad) \& \\
 &)
 \end{aligned}$$

```

(C = exp →
/*L*/   L(T.C) &
/*E*/   (L(T.C) →
          T.C ≡ T &
/*V*/   (V(T.C) ↔ Tm ≠ e))
) &
(C = mvl →
/*L*/   (L(T.C) ↔ Tcl ≠ e)
) &
(C = mvr →
/*L*/   (L(T.C) ↔ Tcr ≠ e)
) &
(C =.mvp →
/*L*/   (L(T.C) ↔ Tm ≠ e) &
/*E*/   (L(T.C) →
          (∀ Tm1)((Tm = Tm1.mvl | Tm = Tm1.mvr) →
                  T.C ≡ T1.Tc.T2.Tm1))
) &
(C = mkl →
/*L*/   (L(T.C) ↔ Tcl = e) &
/*E*/   (L(T.C) →
          T.C ≡ T1.Tcr.T.C.mvp.Tcr.T2.Tm.mvl)
) &
(C = mkr →
/*L*/   (L(T.C) ↔ Tcr = e) &
/*E*/   (L(T.C) →
          T.C ≡ T1.Tcr.Tcl.C.mvp.T2.Tm.mvr)
) &
(C = del →
/*L*/   (L(T.C) ↔ T ≠ e) &
/*E*/   (L(T.C) →
          ((Tm = e → T.C ≡ e) &
           (∀ T11,T22,Tm1)
            (T1 = T11.mkl & T2 =.mvp.T22 & Tm = Tm1.mvl |
             T1 = T11.mkr & T2 =.mvp.T22 & Tm = Tm1.mvr →
              T.C ≡ T11.T22.Tm1))))
)))

```

APPENDIX B

PROTOTYPES FOR SPECIFICATIONS IN APPENDIX A

1. Stack

```
normalform([]).
normalform([push(_) | T]) :- normalform(T).

lp(T, [push(_) ]).
lp(T, [top]) :- append(_, [push(_) ], T).
lp(T, [pop]) :- append(_, [push(_) ], T).

nf(T, [push(S)], Nftr) :- append(T, [push(S)], Nftr).
nf(T, [top], Nftr) :- append([], T, Nftr).
nf(T, [pop], Nftr) :- append(Nftr, [push(_) ], T).

val(T, [top], V) :- append(_, [push(V)], T).
```

2. Queue

```
normalform([]).
normalform([add(_) | T]) :- normalform(T).

lp(T, [add(_) ]).
lp(T, [front]) :- append([add(_) ], _, T).
lp(T, [remove]) :- append([add(_) ], _, T).

nf(T, [add(S)], Nftr) :- append(T, [add(S)], Nftr).
nf(T, [front], Nftr) :- append([], T, Nftr).
nf(T, [remove], Nftr) :- append([add(_) ], Nftr, T).

val(T, [front], V) :- append([add(V)], _, T).
```

3. Overflow Stack

```
normalform([]).
normalform([push(_) | T]) :- normalform(T),
    length(T, L), L < Stacksize.
```

```

lp(T, [push(_)]) :- length(T, L), L < Stacksize.
lp(T, [top]) :- append(_, [push(_)], T).
lp(T, [pop]) :- append(_, [push(_)], T).

nf(T, [push(S)], Nftr) :- append(T, [push(S)], Nftr).
nf(T, [top], Nftr) :- append([], T, Nftr).
nf(T, [pop], Nftr) :- append(Nftr, [push(_)], T).

val(T, [top], V) :- append(_, [push(V)], T).

```

4. Sorting Queue

```

normalform([]).
normalform([add(_)]).
normalform([add(I) | T]) :- append([add(I1)], _, T),
    I =< I1, normalform(T).

lp(T, [add(_)]).
lp(T, [remove]) :- append([add(_)], _, T).
lp(T, [front]) :- append([add(_)], _, T).

nf(T, [add(S)], Nftr) :- parse(S, T, H, R),
    append(H, [add(S)], Temp), append(Temp, R, Nftr).
nf(T, [remove], Nftr) :- append(Nftr, [add(_)], T).
nf(T, [front], Nftr) :- append([], T, Nftr).

val(T, [front], V) :- append(_, [add(V)], T).

parse(I, [], [], []).
parse(I, T, T, []) :- append(_, [add(X)], T), X < I.
parse(I, T, T1, T2) :- append(T1, [add(X)], T), X >= I,
    parse(I, T1, T1, T2), append(T2, [add(X)], T2).

```

5. Table List

```

normalform(T) :- parse(T, _, _, _, _).

lp(T, [insert(_)]).
lp(T, [append(_)]).
lp(T, [delete]) :- parse(T, _, [append(_)], _, _).
lp(T, [goleft]) :- append(T, [goleft], T1), parse(T1, _, _, _, _).
lp(T, [goright]) :- parse(T, _, _, [goleft | _]).
lp(T, [current]) :- parse(T, _, [append(_)], _, _).

nf(T, [insert(S)], Nftr) :- T = [], append([append(S)], [], Nftr), !.
nf(T, [insert(S)], Nftr) :- parse(T, T1, C, T2, G),
    append(T1, [append(S)], T3), append(T3, C, T4),
    append(T4, T2, T5), append(T5, [goleft | G], Nftr).
nf(T, [append(S)], Nftr) :- parse(T, T1, C, T2, G),

```



```

    append(T1,C,T3), append(T3,[append(S)],T4),
    append(T4,T2,T5), append(T5,G,Nftr).
nf(T,[delete],Nftr) :- parse(T,Nftr,C,[],[]).
nf(T,[delete],Nftr) :- parse(T,T1,C,T2,[goleft|G]),
    append(T1,T2,T3), append(T3,G,Nftr).
nf(T,[goleft],Nftr) :- append(T,[goleft],Nftr).
nf(T,[goright],Nftr) :- append(Nftr,[goleft],T).
nf(T,[current],Nftr) :- append(T,[],Nftr).

val(T,[current],V) :- parse(T,_,[append(V)],_,_).

parse([],[],[],[],[]).
parse(T,T1,C,T2,G) :- parse1(T,A,G), parse2(A,G,T1,C,T2).

parse1(T,A,G) :- append(A,G,T), allg(G), alla(A).

parse2(A,G,T1,[C],T2) :- length(A,L1), length(G,L2), L1>L2,
    subtrace(A,L1-L2,Temp1,T2), append(T1,[C],Temp1).

subtrace(T,0,[],T).
subtrace([C|T],I,[C|T1],T2) :- I>0, length(T,L), I=<L+1,
    J is I-1, subtrace(T,J,T1,T2).

allg([]).
allg([goleft|T]) :- allg(T).

alla([]).
alla([append(_) | T]) :- alla(T).

```

6. Unique

```

normalform(T).

lp(T,[getint]).

nf(T,[getint],Nftr) :- append(T,[getint],Nftr).

val(T,[getint],V) :- length(T,V).

```

7. Traversing Stack

```

normalform(T) :- parse(T,_,_,_,_).

lp(T,[push(_)]) :- parse(T,_,_,_,[]).
lp(T,[pop]) :- parse(T,_,[push(_)],_,[]).
lp(T,[down]) :- append(T,[down],Temp), normalform(Temp).
lp(T,[current]) :- parse(T,_,[push(_)],_,_).
lp(T,[to_top]).

```

```

nf(T, [push(X)], Nftr) :- append(T, [push(X)], Nftr).
nf(T, [pop], Nftr) :- append(Nftr, [push(_)], T).
nf(T, [down], Nftr) :- append(T, [down], Nftr).
nf(T, [current], Nftr) :- append(T, [], Nftr).
nf(T, [to_top], Nftr) :- parse1(T, Nftr, _).

val(T, [current], V) :- parse(T, T1, [push(V)], T2, D).

parse([], [], [], [], []).
parse(T, T1, C, T2, D) :- parse1(T, P, D), parse2(P, D, T1, C, T2).

parse1(T, P, D) :- append(P, D, T), alld(D), allp(P).

parse2(P, D, T1, [C], T2) :- length(P, L1), length(D, L2),
    L1 > L2, subtrace(P, L1-L2, Temp1, T2), append(T1, [C], Temp1).

subtrace(T, 0, [], T).
subtrace([C|T], I, [C|T1], T2) :- I > 0, length(T, L), I =< L+1,
    J is I-1, subtrace(T, J, T1, T2).

alld([]).
alld([down|T]) :- alld(T).

allp([]).
allp([push(_)|T]) :- allp(T).

```

8. Priority Queue

```

normalform([]).
normalform([insert(_,_)]).
normalform([insert(P,_)|T]) :- append([insert(P1,_)], _, T),
    P =< P1, normalform(T).

lp(T, [insert(_,_)]).
lp(T, [remove]) :- append([insert(_,_)], _, T).
lp(T, [front]) :- append(_, [insert(_,_)], T).

nf(T, [insert(P, I)], Nftr) :- parse(P, T, T1, T2),
    append(T1, [insert(P, I)], Temp), append(Temp, T2, Nftr).
nf(T, [remove], Nftr) :- append(Nftr, [insert(_,_)], T).
nf(T, [front], Nftr) :- append([], T, Nftr).

val(T, [front], V) :- append(_, [insert(_, V)], T).

parse(P, [], [], []).
parse(P, T, T1, []) :- append(_, [insert(X, _)], T), X < P.
parse(P, T, T1, T2) :- append(T11, [insert(X, Y)], T), X >= P,
    parse(P, T11, T1, T22), append(T22, [insert(X, Y)], T2).

```

9. Shortest Path

```

normalform([]).
normalform(T) :- parse(T, [init(N) | T1], T2),
    legnode(N, T1), legedge(T1, T2).

lp(T, [init(N)]) :- N > 0.
lp(T, [getmaxn]) :- append([init(N)], _, T).
lp(T, [addnode(N1)]) :- append([init(N)], _, T), N1=<N, N1>0.
lp(T, [exnode(N1)]) :- append([init(N)], _, T), N1=<N, N1>0.
lp(T, [delnode(N1)]) :- member(addnode(N1), T).
lp(T, [addedge(S, D, L)]) :- append([init(N)], _, T), not(S=D),
    member(addnode(S), T), member(addnode(D), T), L>=0.
lp(T, [exedge(S, D)]) :- member(addnode(S), T), member(addnode(D), T).
lp(T, [deledge(S, D)]) :- member(addedge(S, D, _), T).
lp(T, [edgelen(S, D)]) :- member(addedge(S, D, _), T).
lp(T, [expath(S, D)]) :- member(addnode(S), T), member(addnode(D), T).
lp(T, [pathlen(S, D)]) :- member(addnode(S), T),
    member(addnode(D), T), val(T, [expath(S, D)], 1).
lp(T, [firststop(S, D)]) :- member(addnode(S), T),
    member(addnode(D), T), val(T, [expath(S, D)], 1).

nf(T, [init(N)], Nftr) :- append([], [init(N)], Nftr).
nf(T, [getmaxn], Nftr) :- append(T, [], Nftr).
nf(T, [addnode(N)], Nftr) :- parse(T, T1, T2),
    delete(addnode(N), T1, T3), append(T3, [addnode(N) | T2], Nftr).
nf(T, [exnode(N)], Nftr) :- append(T, [], Nftr).
nf(T, [delnode(N)], Nftr) :- parse(T, T1, T2),
    delete(addnode(N), T1, T3), delete(addedge(N, _, _), T2, T4),
    delete(addedge(_, N, _), T4, T5), append(T3, T5, Nftr).
nf(T, [addedge(S, D, L)], Nftr) :- parse(T, T1, T2),
    delete(addedge(S, D, _), T2, T3), append(T1, T3, T4),
    append(T4, [addedge(S, D, L)], Nftr).
nf(T, [exedge(S, D)], Nftr) :- append(T, [], Nftr).
nf(T, [deledge(S, D)], Nftr) :- parse(T, T1, T2),
    delete(addedge(S, D, _), T2, T3), append(T1, T3, Nftr).
nf(T, [edgelen(S, D)], Nftr) :- append(T, [], Nftr).
nf(T, [expath(S, D)], Nftr) :- append(T, [], Nftr).
nf(T, [pathlen(S, D)], Nftr) :- append(T, [], Nftr).
nf(T, [firststop(S, D)], Nftr) :- append(T, [], Nftr).

val(T, [getmaxn], V) :- append([init(V)], _, T).
val(T, [exnode(N)], 1) :- member(addnode(N), T).
val(T, [exnode(N)], 0) :- not(member(addnode(N), T)).
val(T, [exedge(S, D)], 1) :- member(addedge(S, D, _), T); S=D.
val(T, [exedge(S, D)], 0) :- not(member(addedge(S, D, _), T)), not(S=D).
val(T, [edgelen(S, S)], 0).
val(T, [edgelen(S, D)], V) :- append(_, [addedge(S, D, V) | _], T).
val(T, [expath(S, S)], 1).
val(T, [expath(S, D)], 1) :- append(_, [addedge(S, D, _) | _], T).
val(T, [expath(S, D)], 1) :- append(_, [addedge(S, X, _) | _], T),

```

```

    val(T, [expath(X,D)], 1), !.
val(T, [expath(S,D)], 0).
val(T, [pathlen(S,S)], 0) :- !.
val(T, [pathlen(S,D)], V) :- append(_, [addedge(S,D1,L)|_], T),
    delete(addedge(S,D1,L), T, T1), val(T1, [expath(S,D)], 1),
    val(T1, [pathlen(S,D)], V1), val(T1, [pathlen(D1,D)], V2),
    min(V1, V2+L, V), !.
val(T, [pathlen(S,D)], V) :- append(_, [addedge(S,D1,L)|_], T),
    delete(addedge(S,D1,L), T, T1), val(T1, [expath(S,D)], 0),
    val(T1, [pathlen(D1,D)], V2), V is V2+L.
val(T, [firststop(V,V)], V) :- !.
val(T, [firststop(S,D)], V) :- val(T, [pathlen(S,D)], V1),
    append(_, [addedge(S,V,L)|_], T),
    val(T, [pathlen(V,D)], V2), V1 is V2+L.

min(A,B,C) :- A<B, !, C is A.
min(A,B,C) :- C is B.

member(X, [X|_]).
member(X, [_|Y]) :- member(X,Y).

delete(X, [], []).
delete(X, [X|Y], Y) :- !.
delete(X, [Y|Z], [Y|R]) :- delete(X,Z,R).

parse([], [], []).
parse(T, [init(N)|T1], T2) :- append([init(N)|T1], T2, T),
    alln(T1), alle(T2).

alln([]).
alln([addnode(_)|T]) :- alln(T).

alle([]).
alle([addedge(_,_,_)|T]) :- alle(T).

legnode(_, []).
legnode(N,T) :- append([addnode(N1)], T1, T), N1=<N, N1>0,
    not(member(addnode(N1), T1)), legnode(N, T1).

legedge(_, []).
legedge(N,T) :- append([addedge(S,D,_)|T1], T),
    member(addnode(S), N), member(addnode(D), N), not(S=D),
    not(member(addedge(S,D,_), T1)), legedge(N, T1).

```

10. Binary Tree

```

normalform(T) :- parse(T,_,_,_,_).

lp(T, [setc(I)]).
lp(T, [exl]).

```



```

lp(T, [exr]).
lp(T, [exp]).
lp(T, [getc]) :- parse(T,_,Tc,_,_), treeparse(Tc,[setc(I)],_,_).
lp(T, [mvl]) :- parse(T,_,Tc,_,_), treeparse(Tc,_,[mkl|_],_).
lp(T, [mvr]) :- parse(T,_,Tc,_,_), treeparse(Tc,_,_,[mkr|_]).
lp(T, [mvp]) :- parse1(T,_,[mvl|_]); parse1(T,_,[mvr|_]).
lp(T, [mkl]) :- parse(T,_,Tc,_,_), treeparse(Tc,_,[],_).
lp(T, [mkr]) :- parse(T,_,Tc,_,_), treeparse(Tc,_,_,[]).
lp(T, [del]) :- T=[setc(I)|_]; T=[mkl|_]; T=[mkr|_].

nf(T, [getc], T).
nf(T, [exl], T).
nf(T, [exr], T).
nf(T, [exp], T).
nf(T, [setc(I)], NfTC) :- parse(T, T1, Tc, T2, Tm),
    treeparse(Tc, Trt, T1, Tr), append3(T1, [setc(I)], T1, Temp1),
    append3(Temp1, Tr, T2, Temp2), append(Temp2, Tm, NfTC).
nf(T, [mvl], NfTC) :- append(T, [mvl], NfTC).
nf(T, [mvr], NfTC) :- append(T, [mvr], NfTC).
nf(T, [mvp], NfTC) :- parse(T, T1, Tc, T2, Tm), append(Tm1, [mvl], Tm), !,
    append3(T1, Tc, T2, Temp), append(Temp, Tm1, NfTC).
nf(T, [mvp], NfTC) :- parse(T, T1, Tc, T2, Tm), append(Tm1, [mvr], Tm),
    append3(T1, Tc, T2, Temp), append(Temp, Tm1, NfTC).
nf(T, [mkl], NfTC) :- parse(T, T1, Tc, T2, Tm), treeparse(Tc, Trt, T1, Tr),
    append3(T1, Trt, [mkl, mvp], Temp1), append3(Temp1, Tr, T2, Temp2),
    append3(Temp2, Tm, [mvl], NfTC).
nf(T, [mkr], NfTC) :- parse(T, T1, Tc, T2, Tm),
    treeparse(Tc, Trt, T1, Tr), append3(T1, Trt, T1, Temp1),
    append3(Temp1, [mkr, mvp], T2, Temp2), append3(Temp2, Tm, [mvr], NfTC).
nf(T, [del], NfTC) :- append(Tt, [mvl], T),
    parse(Tt, T1, Tc, T2, Tm), treeparse(Tc, Trt, T1, Tr),
    append3(T1, Trt, Tr, Temp), append3(Temp, T2, Tm, NfTC).
nf(T, [del], NfTC) :- append(Tt, [mvr], T),
    parse(Tt, T1, Tc, T2, Tm), treeparse(Tc, Trt, T1, Tr),
    append3(T1, Trt, T1, Temp), append3(Temp, T2, Tm, NfTC).
nf(T, [del], NfTC) :- append(_, [mvp], T), NfTC=[].

val(T, [getc], I) :- parse(T,_,Tc,_,_), tel(T,_,[mvr|_]).

val(T, [exp], 0) :- parse1(T,_,[]).

tree([]).
tree(T) :- append(T1, Tr, T), ltree(T1), rtree(Tr).
tree([setc(I)|T]) :- append(T1, Tr, T), ltree(T1), rtree(Tr).

ltree([]).
ltree(T) :- append([mkl|T1], [mvp], T), tree(T1).

```



```

rtree([]).
rtree(T) :- append([mkr|T1],[mvp],T), tree(T1).

match(T,[],[],T,[]).
match(T,[C|Tail],T1,Tc,T2) :- C=mv1,treeparse(T,Trt,T1,Tr),
    append([mkl|T3],[mvp],T1), match(T3,Tail,T4,Tc,T5),
    append3(Trt,[mkl],T4,T1), append3(T5,[mvp],Tr,T2).
match(T,[C|Tail],T1,Tc,T2) :- C=mvr, treeparse(T,Trt,T1,Tr),
    append([mkr|T3],[mvp],Tr), match(T3,Tail,T4,Tc,T5),
    append3(Trt,T1,[mkr],Temp), append(Temp,T4,T1),
    append(T5,[mvp],T2).

treeparse(T,Trt,T1,Tr) :- tree(T), (Trt=[];Trt=[setc(I)]),
    append3(Trt,T1,Tr,T), ltrees(T1), rtree(Tr).

parsel(T,T1,Tm) :- append(T1,Tm,T), tree(T1),allmv1r(Tm).

parse(T,T1,Tc,T2,Tm) :- parsel(T,Tt,Tm), match(Tt,Tm,T1,Tc,T2).

allmv1r([]).
allmv1r([T1|T]) :- (T1=mv1;T1=mvr),allmv1r(T).

append3(A,B,C,D) :- append(A,B,X),append(X,C,D).

```

VITA

Surname: Wang

Given Names: Yabo

Place of Birth: Bengbu, Anhui, China

Date of Birth: September 26, 1958

Educational Institutions Attended, with Dates of Entering and Leaving:

University of Science & Technology of China, Hefei, Anhui, China 1978 to 1982

University of Victoria, Victoria, B.C. Canada 1986 to 1987

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B.Sc. University of Science and Technology of China, China 1982

Honors and Awards:

University of Victoria Fellowship 1986-1987 1987-1988

Publications:

- (1) **Microcomputer Emulation Strategies**, First Chinese Conference on Firmware, Shanghai, P.R.China, December 1983
- (2) **Executable Prototypes of Trace Specifications**, (with D. Hoffman), Canadian Information Processing Societe Conference, Edmonton, Canada, November 1987

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make *single copies only* for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Prototypes of Trace Specifications

Author



Yabo Wang

April 21, 1988

Date