SOME TERMINOLOGY FOR THIS TALK

1. MODULE
   A set of programs intended as a cohesive work assignment
2. PACKAGE
   A set of programs working on a COMPLETELY private data structure
3. INFORMATION HIDING PRINCIPLE
   Modules should be packages!
   Things likely to change should be private.
   Interface should abstract from "secrets".
4. OBJECT/VARIABLE
   a. Finite state machine.
   b. Independent of all other objects.
   c. Viewed as a collection of access programs.
   d. May be grouped into "types" in arbitrary ways.
   e. Many objects may be implemented by a single module.
   f. We neglect resource interactions.

What are "specifications"?

A. General definition of specification

   Specific information about the object

B. Engineering definition

   Specific information about the requirements the object must meet

   Must be "black-box" descriptions

   Internal design decisions are not requirements

   Properties such as "ease of change" are requirements

C. We will use it in the engineering sense

Why do we need module specifications?

A. Multiperson projects.

B. Multiversion projects.

C. "Our inability to do much" (E. W. Dijkstra).

Each subtask should have a definition independent of the rest of the job.

D. Making early decisions explicit and precise.

1. Intramodule assumptions.
2. Decision postponement.

Why must specifications be precise?

A. Early, distributed design decisions are hard to correct.

B. Prevent incompatibility between parts.

C. Remove the need for excessive information distribution.

- Napkin Story

D. Minimise forbidden assumptions.

Why must specifications be abstract?

A. Abstraction--one model, many realisations.

B. Must allow many versions.

C. State only requirements e.g.: fictitious sort.

D. Less information to comprehend.

E. User only concerned about that which he could eventually discover for himself by legitimate use.

What do we mean by formal?

A. Not "superficial"

B. Not full of greek letters, subscripts, and superscripts.

C. Based on restricted forms and strict interpretation rules.

  1. Reduced chance of misinterpretation.

  2. Mechanically interpretable.

Why not Natural Language (French, German, English, Dutch, ...?)

A. Interpretation requires an elaborate legal system.

B. Examples of subtle ambiguities in natural language specifications.

1. Delivers the top of the stack.
2. Delivers the address of the new PSW.
3. Removes the top element from the stack.
4. The date three months from today.

Module Specifications Vs. Program Specifications

Programs do not hide data.

Program effects can be described in terms of data structure.

Program effects are visible immediately.

Modules have hidden data.

Module specifications may not mention the data structure.

Module effects can have delayed visibility.

We use relational specifications for programs.

We use "trace assertions" for modules.

The two are completely compatible.

The basic rules of abstract formal specifications

A. Stating the visible effects of module access programs on each other.

B. Refusal to mention internal or invisible effects.

The road to abstract specifications.

C. Leaving some externally visible values undefined.

To restrict statements to requirements.

Syntax in a Specification

A. What is a type?

An equivalence class of variables.

There are <u>many</u> equivalence relations of interest.

A variable may be of more than one type.

B. Syntax section presents type information.

Defines the permitted set of variables/values as parameters.

Define where the return value may be used.

Stating the "syntactic" properties of module access programs

A. Does the module access program have a value? What is its type?

B. Input parameters: How many? What type?

C. Output parameters: How many? What type?

D. This information specifies the syntactically allowed invocations in a program text.

E. More information can be added by defining more types.

The line between syntax and semantics can be moved.

Describing "don't cares", three approaches:

A. Leave certain values undefined

- may be mistaken for incompleteness

B. Use a special symbol for "undefined"

- makes the specification larger, but checkable

C. Describe the set of possible values (non-determinism)

Specifying forbidden actions, three approaches

A. State the allowed actions, ignore the others.

B. State the effects of restriction violation
(if checking affordable).

C. Use non-determinism.

COMMUNICATION WITH OBJECTS

1. Input variables: object observes these "continuously".

2. Output variables:

   the only information available from an object.

   information available continuously

3. Input arguments of access programs.

4. Output arguments of access programs.

Content vs. Notation

A. Rules about content important.

B. Syntactic invention still needed.

C. Don't let syntax control content.

The Structure Of Specifications

Mathematical generality is nice, but

It does not always help,

A rigid structure makes documents easier to check,

A rigid structure makes specifications easier to write.

What is a Trace?

A. Execution history of a module from creation

All events affecting the module

Usually invocations of access programs

B. A subtrace is part of a trace, not necessarily the initial part

We make assertions about traces, not subtraces

C. Notation for describing traces and subtraces

1. PUSH(a)
2. PUSH(A).Y(4)
3. _
4. $[\text{TOP.PUSH}(a_i)]_{i=1}^{n}$
5. output values can appear in traces

Why are Traces Important?

Any property of the module that concerns the user is visible.

Any visible property can be expressed in terms of traces and returned values.

If we can define the set of allowed traces, we have defined the circumstances under which the module must function.

If we define returned values as a function of the history (trace), we have specified what the module must do.

When can two traces be equivalent? (≡)

Equivalent traces must be indistinguishable from outside the module.

What is a canonical trace?

There is an infinite set of possible traces.

Because the module is a finite state machine, there are equivalent traces.

The equivalence relation determines a finite set of equivalence classes.

We pick one representative from each class as canonical.

Why are canonical traces important?

1. They provide a simplified way of defining the equivalence relation.

2. We can simplify the description of certain functions by restricting the domain to canonical traces.

McMaster University

Extensions of a Canonical Trace by a Single Event

A) The result may be a canonical trace.

B) The result may be equivalent to a canonical trace.

A trace assertion specification consists of:

1) A syntax section naming the access programs and the types of the parameters and return values.

2) A predicate defining the canonical form for traces.

3) A set of functions, one corresponding to each access program, specifying the canonical trace equivalent to an extension of a canonical trace by an invocation of that access program.

   These functions also define the legality of the extension. The range is (newtrace,%class%). %class% provides information for error recovery.

4) A set of relations/functions defining the set of allowable values to be returned for a given canonical trace.

5) Functions defining return values in terms of output values.

When is a specification complete?

If the domain of each extension function includes all canonical traces, and all possible extensions.

If the domain of the relations defining values includes all canonical traces.

When is a specification consistent?

When all equivalent traces have the same set of allowed values,

When the extension functions are functions.

McMaster University

How Do We Handle Non-Determinism

1. For non-deterministic cases the values should be included in the trace.

2. We still use extension functions, a unique canonical trace. *

3. Value functions can be relations.

* this is a debateable decision.

Communications Research Laboratory
Software Engineering Research Group
"*connecting theory with practice*"

Choice vs. Non-determinism

1. Non-determinism is one form of “don't care”.
2. Although we may not care about the answer, we may want consistency.
3. Then we must provide a set of deterministic specifications.

The Use Of Tables As Expressions

1. The rows partition the domain.
2. The expressions defining the values are simplified.
3. Systematic checks for completeness and consistency are easier.
4. Checks for correctness are easier.

Existential Quantification Of Free Variables

1. Full quantification clutters up expressions.

2. Automatic existential quantification of free variables gives a “pattern match” semantics.

3. We use default duplication of left hand side definitions.

Naming Objects

1. Multiple-object modules require the ability to name objects.

2. Both names and values may appear in traces.

Traces Used As Values

1. We need a notation for "literal" values of abstract types.
2. The canonical trace serves that role.
3. Shorthand can be defined in terms of canonical traces.
4. These values can be used in our relational program semantics.
5. Objects, are data elements for higher level programs.

What is the effect of programming languages on specifications

A. Lack of choices in some languages leads to simplification.

B. Lack of "functions" leads to minor complication.

C. User-defined types can simplify specifications.

D. Scope rules can compromise implementations.

FIGURE 1: UNBOUNDED INTEGER STACK MODULE

TYPE IMPLEMENTED: <stack>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
top	<integer>

ACCESS PROGRAMS

Program Name	Value	Arg#1
PUSH		<integer>
POP		
TOP	<integer>	

(2) CANONICAL TRACES

canonical(T) $\leftrightarrow$ (T = $[PUSH(a_i)]_{i=1}^{n}$)

(3) EQUIVALENCES

T.PUSH(a) ≡ T.PUSH(a)

T.POP ≡

conditions	equivalences
T = _	%empty%
T ≠ _	T1 where T = T1.PUSH(a)

T.TOP ≡

conditions	equivalences
T = _	%empty%
T ≠ _	T

McMaster University

(4) OUTPUT VALUES

V[top](T) =

conditions	equivalences
T = _	%undefined%
T ≠ _	a where T = T1.PUSH(a)

(5) RETURN VALUES

Program Name	Argument No	Value
TOP	Value	top

FIGURE 2: MULTIPLE UNBOUNDED INTEGER STACK MODULE

TYPE IMPLEMENTED: <stack>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
top	<integer>

ACCESS PROGRAMS

Program Name	Value	Arg#1	Arg#2
CREATESTACK		<name>:0	
PUSH		<name,stack>:0	<integer>
POP		<name,stack>:0	
TOP	<integer>	<stack>	

(2) CANONICAL TRACES

$$\text{canonical}(T_s) \leftrightarrow (T_s = \_\,) \vee (T_s = \text{CREATESTACK}(\text{'S'}).[\text{PUSH}(<\text{'S'},\$>,a_i)]_{i=1}^{n}$$

(3) EQUIVALENCES

T_s.CREATESTACK('S') ≡ CREATESTACK('S')

T_s.PUSH(<'S',$>,a) ≡

conditions	equivalences
T = _	%invalidstack%
T ≠ _	T_s.PUSH(<'S',$>,a)

T_s.POP(<'S',$>,a) ≡

conditions	equivalences
T_s = _	%invalidstack%
length(T_s) = 1	%empty%
length(T_s) > 1	$T1_s$ where T_s = $T1_s$.PUSH(<'S',$>,a)

(4) VALUES

OUTPUT VALUES

V[top](T_s) =

conditions	equivalences
length(T_s) ≤ 1	%undefined%
length(T_s) > 1	a where T_s = $T1_s$.PUSH(<'S',$>,a)

RETURN VALUES

Program Name	Argument No	Value
TOP	Value	top

FIGURE 3: INTEGER QUEUE WITH CAPACITY = 12

TYPE IMPLEMENTED: <queue>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
front	<integer>

ACCESS PROGRAMS

Program Name	Value	Arg#1
ADD		<integer>
REMOVE		
FRONT	<integer>	

(2) CANONICAL TRACES

canonical(T) $\leftrightarrow$ (T = $[ADD(a_i)]_{i=1}^{n}$) $\wedge$ $(0 \leq n \leq 12)$

(3) EQUIVALENCES

T.ADD(a) ≡

conditions	equivalences
length(T) = 12	%full%
length(T) < 12	T.ADD(a)

T.REMOVE ≡

conditions	equivalences
T = _	%empty%
T ≠ _	S1 where T = ADD(a) . S1

T.FRONT ≡

conditions	equivalences
T = _	%empty%
T ≠ _	T

(4) VALUES

OUTPUT VALUES

V[front](T) =

conditions	equivalences
T = _	undefined
T ≠ _	S1 where T = ADD(a) . S1

RETURN VALUES

Program Name	Argument No	Value
FRONT	Value	front

FIGURE 4: OVERFLOW INTEGER STACK MODULE

TYPE IMPLEMENTED: <stac>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
top	<integer>

ACCESS PROGRAMS

Program Name	Value	Arg#1
PUSH		<integer>
POP		
TOP	<integer>	

(2) CANONICAL TRACES

canonical(T) $\leftrightarrow$ (T = $[(PUSH)(a_i)]_{i=1}^{n}$) $\wedge$ (n $\leq$ #stacksize#)

DICTIONARY

#stacksize#: specification parameter

(3) EQUIVALENCES

T.PUSH(a) $\equiv$

conditions	equivalences
length(T) < #stacksize#	T.PUSH(a)
length(T) = #stacksize#	S1.PUSH(a) where T = PUSH(b).S1

T.POP $\equiv$

conditions	equivalences
T = _	%empty%
T $\neq$ _	T | where T = T1 PUSH(a)

T.TOP $\equiv$

conditions	equivalences
T = _	%empty%
T $\neq$ _	T

(4) VALUES

OUTPUT VALUES

V[top](T) =

conditions	equivalences
T = _	%undefined%
T $\neq$ _	a mod 255 where T = T1.PUSH(a)

RETURN VALUES

Program Name	Argument No	Value
TOP	Value	top

FIGURE 4 A: OVERFLOW INTEGER STACK MODULE (Version 2)

TYPE IMPLEMENTED: <stac>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
top	<integer>

ACCESS PROGRAMS

Program Name	Value	Arg#1
PUSH		<integer>
POP		
TOP	<integer>	

(2) CANONICAL TRACES

canonical(T) $\leftrightarrow$
(T = $[(PUSH)(a_i)]_{i=1}^{n}$) $\wedge$ (n $\leq$ #stacksize#) $\wedge$
($\forall$ a_i , 0 $\leq$ a_i $\wedge$ $a_i \leq 254$)

DICTIONARY

#stacksize#: specification parameter

(3) EQUIVALENCES

T.PUSH(a) $\equiv$

conditions	equivalences
length(T) < #stacksize#	T.PUSH(a mod 255)
length(T) = #stacksize#	S1.PUSH(a mod 255) where T = PUSH(b).S1

■ McMaster University ■

T.POP $\equiv$

conditions	equivalences
T = _	%empty%
T $\neq$ _	T1 where T = T1.PUSH(a)

T.TOP $\equiv$

conditions	equivalences
T = _	%empty%
T $\neq$ _	T

(4) VALUES

OUTPUT VALUES

V[top](T) =

conditions	equivalences
T = _	%undefined%
T $\neq$ _	a where T = T1.PUSH(a)

RETURN VALUES

Program Name	Argument No	Value
TOP	Value	top

FIGURE 5: UNBOUNDED INTEGER TABLE LIST MODULE

TYPE IMPLEMENTED: <T/L>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
left	<boolean>
right	<boolean>
out	<boolean>
current	<integer>

ACCESS PROGRAMS

Program Name	Value	Arg#1
INSERT		<integer>
ALTER		<integer>
DELETE		
EXLEFT	<boolean>	
EXRIGHT	<boolean>	
OUT	<boolean>	
GOLEFT		
GORIGHT		
CURRENT	<integer>	

(2) CANONICAL TRACES

canonical(T) $\leftrightarrow$ (T = $[INSERT(a_i)]_{i=1}^{m} . [GOLEFT]_{i=1}^{n}$) $\wedge$ (m $\geq$ n)

DICTIONARY

AUXILIARY FUNCTIONS

Function Name	Value	Arg#1	Arg#2	Arg#3	Arg#4	Arg#5
parse	<boolean>	<trace>	<trace>	<trace>	<trace>	<trace>

parse(S,S1,C,S2,G) =

conditions	equivalences
(S = S1.C.S2.G) $\wedge$ (S1 = $[INSERT(a_i)]_{i=1}^{m}$) $\wedge$ (C = INSERT(a)) $\wedge$ (S2 = $[INSERT(b_i)]_{i=1}^{n}$) $\wedge$ (G = $[GOLEFT]_{i=1}^{n}$)	true
else	false

(3) EQUIVALENCES

T.INSERT(a) ≡

conditions	equivalences
count(T,INSERT) = count(T,GOLEFT)	INSERT(a).T
count(T,INSERT) > count(T,GOLEFT)	T1.C.INSERT(a).S1.G where parse(T,T1,C,S1,G)

T.ALTER(b) ≡

conditions	equivalences
count(T,INSERT) = count(T,GOLEFT)	%noncurrent%
count(T,INSERT) > count(T,GOLEFT)	T1.INSERT(b).S1.G where parse(T,T1,INSERT(a),S1,G)

T.DELETE ≡

conditions	equivalences
count(T,INSERT) = count(T,GOLEFT)	%noncurrent%
count(T,INSERT) > count(T,GOLEFT)	T1.S1.G where parse(T,T1,C,S1,G)

T.EXLEFT ≡ T

T.EXRIGHT ≡ T

T.OUT ≡ T

T.GOLEFT ≡

conditions	equivalences
count(T,INSERT) = count(T,GOLEFT)	%noleft%
count(T,INSERT) > count(T,GOLEFT)	T.GOLEFT

T.GORIGHT ≡

conditions	equivalences
count(T,GOLEFT) = 0	%noright%
count(T,GOLEFT) > 0	T1 where T = T1.GOLEFT

T.CURRENT ≡

conditions	equivalences
count(T,INSERT) = count(T,GOLEFT)	%noncurrent%
count(T,INSERT) > count(T,GOLEFT)	T

(4) VALUES

OUTPUT VALUES

V[list](T) =

conditions	value
count(T,INSERT) - count(T,GOLEFT) > 1	true
count(T,INSERT) - count(T,GOLEFT) ≤ 1	false

McMaster University

V[right](T) =

conditions	value
count(T,GOLEFT) > 0	true
count(T,GOLEFT) = 0	false

V[out](T) =

conditions	value
count(T,INSERT) = count(T,GOLEFT)	true
count(T,INSERT) > count(T,GOLEFT)	false

V[current](T) =

conditions	value
count(T,INSERT) = count(T,GOLEFT)	%undefined%
count(T,INSERT) > count(T,GOLEFT)	a where parse(T,T1,INSERT(a),S1,G)

RETURN VALUES

Program Name	Argument No	Value
EXLEFT	Value	left
EXRIGHT	Value	right
OUT	Value	out
CURRENT	Value	current

FIGURE 6: UNBOUNDED UNIQUE INTEGER PRODUCER MODULE

TYPE IMPLEMENTED: <int>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
top	<integer>

ACCESS PROGRAMS

Program Name	Value
GETINT	<integer>

(2) CANONICAL TRACES

canonical(T) $\leftrightarrow$
$(T = [\ (x_{i-1})\ .\ GETINT\ ]_{i=1}^{n}) \wedge$
$(\forall\ T1,S1,S2,x_i,x_j)((T = T1.(x_i).S1.(x_j).S2) \rightarrow (x_i < x_j))$

(3) EQUIVALENCES

T.VA[T].GETINT $\equiv$ T1.VA[T].GETINT.S1
where
(T = T1.S1) $\wedge$ canonical(T1.VA[T].GETINT.S1)

(4) VALUES

OUTPUT VALUES

V[int](T) $\in$ { a | $\neg$ $\exists$ T1,S1)(T = T1.(a).S1) }

RETURN VALUES

Program Name	Argument No	Value
GETINT	Value	top

FIGURE 7: UNBOUNDED PRIORITY INTEGER QUEUE MODULE

TYPE IMPLEMENTED: <pqueue>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
front	<integer>

ACCESS PROGRAMS

Program Name	Value	Arg#1	Arg#2
INSERT		<integer>	<integer>
REMOVE			
FRONT	<integer>		

(2) CANONICAL TRACES

canonical(T) $\leftrightarrow$
$(T = [\ (x_{i-1})\ .\ INSERT(p_i, x_i)\ ]_{i=1}^{n}\ ) \wedge (\ (x_{0i})\ ) = (\%empty\%)) \wedge$
$(\forall\ T1,S1,p,p1,x,x1)((T = T1.INSERT(p,x).(x).INSERT(p1,x1).S1 \rightarrow$
$(p < p1) \vee ((p = p1) \wedge (x \leq x1))$

(3) EQUIVALENCES

T.VA[T].INSERT(a,b) $\equiv$

conditions	equivalences
T = _	T.VA[T].INSERT(a,b)
(T = T1.INSERT(p,x)) $\wedge$ ((p > a) $\vee$ (p = a) $\wedge$ (x > b))	S1.INSERT(a,b).(b).S2 where (T = S1.S2) $\wedge$ canonical(S1.INSERT(a,b).(b).S2)
(T = T1.INSERT(p,x)) $\wedge$ ((p < a) $\vee$ (p = a) $\wedge$ (x $\leq$ b))	T1.INSERT(p,x).(x).INSERT(a,b)

T.VA[T].REMOVE $\equiv$

conditions	equivalences
T = _	%empty%
T = T1.(y).INSERT(p,VA[T])	T1
T = T1.INSERT(p,VA[T]).VA[T].S1) $\wedge$ ($\forall$ S2,S3,u,v)((S1= S2.INSERT(u,v).S3 $\rightarrow$ (u = p $\wedge$ (v > VA[T]))	T1.S1

T.VA[T].FRONT $\equiv$ T

(4) VALUES

OUTPUT VALUES

V[front](T,O)

conditions	values
T = _	%empty%
T $\neq$ _	O = a where (T = T1.INSERT(p,a).S1) $\wedge$ ($\forall$ S2,S3,u,v)((T = S2.INSERT(u,v).S3 $\rightarrow$ (u $\leq$ p))

RETURN VALUES

Program Name	Argument No	Value
FRONT	Value	front

FIGURE 8: MULTIPLE BINARY INTEGER TREE MODULE

TYPE IMPLEMENTED: <tree>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
tree	<tree>

ACCESS PROGRAMS

Program Name	Value	Arg#1	Arg#2	Arg#3	Arg#4
SETNIL		<name>:0			
SETTREE		<name>:0	<integer>	<tree>	<tree>
ALTERNODE		<name,tree>:0	<path>	<integer>	
ALTERTREE		<name,tree>:0	<path>	<tree>	
GETNODE	<integer>	<tree>	<path>		

(2) CANONICAL TRACES

canonical(T_R) $\leftrightarrow$

(T_R = _) $\vee$
(T_R = SETNIL('R')) $\vee$
(T_R = SETTREE('R',i,U,V))

DICTIONARY

EXTERNAL TYPES

<path> PATH HOLDER MODULE

AUXILIARY FUNCTIONS

Program Name	Value	Arg#1	Arg#2	Arg#3
validpath	<boolean>	<tree>	<path>	
update	<tree>	<tree>	<path>	<tree>
change	<tree>	<tree>	<path>	<integer>
getval	<integer>	<tree>	<path>	

validpath(R,P) =

conditions	values
(T_R = SETTREE('R',i,U,V)) ∧ (EMPTYPATH(P) ∨ GOLEFT(P) ∧ validpath(U,SUBPATH(P)) ∨ GORIGHT(P) ∧ validpath(V,SUBPATH(P)))	true
else	false

update(R,P,X) =

conditions	values
validpath(R,P) ∧ EMPTYPATH(P) ∧ (T_X = SETNIL('X'))	SETNIL('R')
validpath(R,P) ∧ EMPTYPATH(P) ∧ (T_X = SETTREE('X',i,U,V))	SETTREE('X',i,U,V)
validpath(R,P) ∧ GOLEFT(P) ∧ (T_R = SETTREE('R',i,U,V))	SETTREE('R',i,update(U,SUBPATH(P),X),V)
validpath(R,P) ∧ GORIGHT(P) ∧ (T_R = SETTREE('R',i,U,V))	SETTREE('R',i,U,update(V,SUBPATH(P),X))

change(R,P,j) =

conditions	values
validpath(R,P) ∧ EMPTYPATH(P)	SETTREE('R',j,U,V) where
else	false

update(R,P) =

conditions	values
validpath(R,P) ∧ EMPTYPATH(P)	i where T_R = SETTREE('R',i,U,V)
validpath(R,P) ∧ GOLEFT(P)	getval(U,SUBPAHT(P)) where T_R = SETTREE('R',i,U,V)
validpath(R,P) ∧ GORIGHT(P)	getval(V,SUBPAHT(P)) where T_R = SETTREE('R',i,U,V)

(3) EQUIVALENCES

(T_R.SETNIL('R') ≡ SETNIL('R')

T_R.SETTREE('R',i,U,V) ≡ SETTREE('R',i,U,V)

T_U.SETTREE('R',i,U,V) ≡

conditions	values
'R' = 'U'	SETTREE('U',j,U,V)
'R' ≠ 'U'	T_U

T_V.SETTREE('R',i,U,V) ≡

conditions	values
'R' = 'V'	SETTREE('V',j,U,V)
'R' ≠ 'V'	T_V

T_R.ALTERTREE(<'R',$>,P,X) ≡

conditions	values
¬validpath(R,P)	%invalidpath%
validpath(R,P)	update(R,P,X)

T_X.ALTERTREE(<'R',$>,P,X) ≡

conditions	equivalences
¬validpath(R,P)	%invalidpath%
validpath(R,P) ∧ ('X' = 'R')	update(X,P,X)
validpath(R,P) ∧ ('X' ≠ 'R')	T_X

T_R.ALTERNODE(<'R',$>,P,i) ≡

conditions	equivalences
¬validpath(R,P)	%invalidpath%
validpath(R,P)	change(R,P,i)

T_R.GETTREE(<'R',$>,P,X) ≡

conditions	equivalences
¬validpath(R,P)	%invalidpath%
validpath(R,P) ∧ (T_R = SETNIL('R'))	%empty%
validpath(R,P) ∧ (T_R ≠ SETNIL('R'))	T_R

(4) VALUES

OUTPUT VALUES

V[tree](T_R) = T_R

RETURN VALUES

Program Name	Argument No	Value	
GETNODE	Value	¬validpath(Arg#1,Arg#2)	%invalidpath%
		validpath(Arg#1,Arg#2)	getval(Arg#1,Arg#2)

FIGURE 9: PATH HOLDER MODULE

TYPE IMPLEMENTED: <path>

(1) SYNTAX

OUTPUT VARIABLES

Variable Name	Type
empty	<boolean>
left	<boolean>
right	<boolean>

ACCESS PROGRAMS

Program Name	Value	Arg#1	Arg#2
MAKEPATH		<name>:0	
ADDPATH		<name,path>:0	<char>
SUBPATH		<name,path>:0	
EMPTYPATH	<boolean>	<path>	
GOLEFT	<boolean>	<path>	
GORIGHT	<boolean>	<path>	

(2) CANONICAL TRACES

$$\text{canonical}(T_R) \leftrightarrow (T_P = \_\ ) \vee (T_P = \text{MAKEPATH('P')}.[\text{ADDPATH}(<\text{'P'},\$>,c_i)]_{i=1}^{n})$$

(3) EQUIVALENCES

T_P.MAKEPATH('P') ≡ MAKEPATH('P')

T_P.EMPTYPATH($) ≡

conditions	values
$T_P = \_$	%invalidpath%
$T_P \neq \_$	T_P

T_P.GOLEFT($) ≡

conditions	equivalences
$T_P = \_$	%invalidpath%
$T_P \neq \_$	T_P

T_P.GORIGHT($) ≡

conditions	equivalences
$T_P = \_$	%invalidpath%
$T_P \neq \_$	T_P

T_P.ADDPATH(<'P',$>,c) ≡

conditions	equivalences
$T_P = \_$	%invalidpath%
$(T_P \neq \_) \wedge (c \neq \text{'l'}) \wedge (c \neq \text{'r'})$	%wrongdirection
$(T_P \neq \_) \wedge ((c \neq \text{'l'}) \vee (c \neq \text{'r'}))$	T_P.ADDPATH(<'P',$>,c)

T_P.SUBPATH(<'P',$>) ≡

conditions	equivalences
$T_P = \_$	%invalidpath%
T_P = MAKEPATH('P')	%emptypath%
T_P = MAKEPATH('P').ADDPATH(<'P',$>,c).$T1_P$	T_P.ADDPATH(<'P',$>,c)

(4) VALUES

OUTPUT VALUES

V[empty](T_P) =

conditions	equivalences
T_P = _	%undefined%
T_P = MAKEPATH('P')	true
T_P = MAKEPATH('P').ADDPATH(<'P',$>,c).$T1_P$	false

V[left](T_P) =

conditions	equivalences
T_P = _	%undefined%
T_P = MAKEPATH('P').ADDPATH(<'P',$>,'1').$T1_P$	true
else	false

V[right](T_P) =

conditions	equivalences
T_P = _	%undefined%
T_P = MAKEPATH('P').ADDPATH(<'P',$>,'r').$T1_P$	true
else	false

RETURN VALUESS

Program Name	Argument No	Value
EMPTYPATH	Value	empty
GOLEFT	Value	left
GORIGHT	Value	right