On Trace Specifications

Theodore S. Norvell[1]
1995 July 10

Abstract: In this report I explore some ideas for formally specifying modules based on the trace assertion method outlined in, for example, [Parnas and Wang 1989].

These ideas include:

- A formal mathematical theory of trace specifications which is independent of their intended application to module specification (Chapter 2).
- Some ideas on presenting module specifications (Chapter 3).
- A theory of trace specifications for dealing with modules that call other modules (Chapter 4).
- Automata theoretic models for trace specifications of the sort defined in Chapter 4 (Chapter 5).
- A theory of trace specifications for cases where deterministic automata are not suitable models (Chapter 6).

1. Author's address: Faculty of Engineering and Applied Science, Memorial University of Newfoundland, St.John's NFLD, A1C 5S7, Canada. Email: norvell@mcserg.crl.mcmaster.ca

Contents:

CHAPTER 1 Introduction and Notation -- 6
1.1 Introduction..6
1.2 Some Notational Matters 7
1.2.1 Segments and Sequences 7
1.2.2 Variable Binding Constructs 8
1.2.3 Cartesian Products 9
1.2.4 Tables 9
1.2.5 Try-else 9
1.2.6 Discussion 10

CHAPTER 2 Theory of Leaf Modules -- 11
2.1 Introduction...11
2.2 Basic Definitions..11
2.3 A Few Theorems About Trace Specifications 12
2.3.1 Support for the theorems...15
2.4 Determinism..15
2.5 Discussion 15

CHAPTER 3 Presentation of Leaf Module Specifications -------------------------------- 18
3.1 Trace specification documents...18
3.1.1 An Example 18
3.1.2 Pattern Matching..20
3.1.3 Using Vector Equality Tables...21
3.1.4 Discussion 21
3.2 Nondeterminism...22
3.2.1 Independent Nondeterminism...22
3.2.2 Dependent Nondeterminism 23
3.3 Conclusions...24
3.3.1 Conciseness and Clarity. 24
3.3.2 Comparison with [Wang 1994] and [Parnas and Wang 1989]..........................25

CHAPTER 4 Theory and Presentation of Nonleaf Module Specifications-------------- 27
4.1 Introduction...27
4.2 Running Example 28
4.3 Two-faced Trace Specification Theory 29
4.4 Presentations 31

4.4.1 A Process-Model Specification....31
4.4.2 A Variable-Model Specification....32

CHAPTER 5 Automata Theoretic Models---- 34
5.1 Dichromatic Automata....34
5.2 Trichromatic Automata....35

CHAPTER 6 Exotic Nondeterminism---- 38
6.1 An Example....38
6.2 Definitions....38
6.3 Theorems....39
6.4 Presentation....41

CHAPTER 7 Concluding Remarks---- 43
7.1 Further Work....43
7.1.1 Nonleaf Modules....43
7.1.2 Automata Theory of Exotic Nondeterminism....43
7.1.3 Data refinement....43
7.1.4 Multiple Objects....43
7.2 Acknowledgments....43

CHAPTER 8 **Bibliography**---- 44

Index of defined terms:

A
alphabet 29

B
black state 34
bottom alphabet 29

C
canonical 11, 29
canonical trace 11
catenation 7
competence function 11, 29
compose 7

D
dependent nondeterminism 23
dichromatic automaton 34

E
equivalence 12, 14
event 11
event class 18
event-response pair 11
exotic nondeterminism 38
exotic pre-trace specification 38
exotic trace specification 39
expanded competence function 12, 31, 39
expanded output function 12, 30, 39
extension 14, 41
extension function 11, 29

F
feasible trace 11
finite sequence 7
firewall interpretation 17

G
green state 34

I
independent nondeterminism 23
initial trace 11, 29

L
lambda expression 8
LD interpretation 16
length 7

M
module-as-process 27
module-as-variable 27

N
nondeterminism, dependant 23
nondeterminism, exotic 38
nondeterminism, independent 23
normal function table 9

O
observational equivalence 14, 41
output 11
output function 11, 29

P
pre-trace specification 11
pre-two-faced trace specification 29

Q
quantified formula 8
quantified term 8

R
red state 35
reduction function 11
response 11
response class 19

S
segment 7
sequence 7
sequence comprehension 9
simple trace 29
solution 8
solution comprehension 8
specification equivalence 12

T
trace 12
trace specification 12
trichromatic automata 35
two-faced trace specification 30

V
VDM interpretation 16
vector equality table 9

CHAPTER 1 Introduction and Notation

1.1 Introduction

The trace assertion method should be set on firm mathematical foundations and presentations of trace specifications should use a minimum of mathematical notation peculiar to trace specifications. This short report presents some ideas for achieving these goals. It builds on the ideas in [Wang 1994], [Parnas and Wang 1989], and [Iglewski, Madey, and Stencel 1994]. Although this report is intended to be self contained, readers not familiar with the earlier work on trace specifications will likely find the introductory material a little brisk.

The structure of this report is a theme and variations. The theme is set in Chapter 2 with a simple mathematical theory of trace specifications. From there the reader can explore the other chapters in virtually any order.

The reader interested in presentations of trace specifications will go to Chapter 3.

The reader interested in the specifications of modules that use other modules may want to skip to Chapter 4, although the presentations used as examples depend on Chapter 3.

The reader interested in automata theoretic models can skip to the first part of Chapter 5. The second part presents models for the theory in Chapter 4.

The reader interested in further exploring nondeterminism may want to skip straight to Chapter 6, although the example presentation depends on Chapter 3.

The original motivation for this report came from teaching trace specifications to third year undergraduate engineering students. I found that the mathematical notations used in published descriptions of the trace assertion method were too particular to the trace specifications. Instead of introducing new notations, I wanted to teach the method. Luckily I found that the notations the students had already been taught — standard notations from predicate calculus and set theory — together with simple tables were quite adequate and, in some ways, superior for expressing trace specifications.

The improvements, if improvements they be, are two-fold. First the underlying theory is slightly different from that in previous work, and is spelled out in the form of an axiomatic theory. Second there are cosmetic changes to the presentation of trace specifications.

The value of trace specifications is not just that they form a nice theory, but that they can actually be used to document programs at the locations where documentation can be most effective: i.e. at module interfaces. It is therefore important that trace specifications be as readable as possible. So any changes to the presentation of trace specifications that enhances their readability are desirable.

The main changes to the theory are an extension to cover cases where the empty trace is not canonical and a new way of expressing misuse of modules. The theory here, as in [Iglewski, Madey, and Stencel 1994] and [Janicki 1995], and in contrast to [Wang 1994] and [Parnas and Wang 1989], treats nondeterminism by putting the selection of responses prior to the selection of a next canonical trace.

The main changes to the presentation of trace specifications that I propose are (a) the use of simple tables with no odd rules about quantification and (b) the reduction of pointless repetition. There are also minor notational clarifications. For example, trace specifications, as presented in earlier papers, have used variables which are implicitly quantified over limited scopes. Such notational quirks all contribute to making trace specifications a little mysterious to the uninitiated.

1.2 Some Notational Matters

The examples and discussion that follow will require some notation for mathematical objects and operations. Therefore, this section introduces that notation that is not entirely standard in mathematics. Most of the particular notations introduced are not closely tied to the other aspects of this paper, so if you like them, use them, if you don't, then please bear with me.

1.2.1 Segments and Sequences

Segments of the integer numbers are often useful, so I will use the abbreviation:

$$\{i, ...j\} = \{k \in ints : i \le k < j\} \quad .$$

Note the asymmetry. A function with domain $\{0, ...n\}$ (for some natural n) is called a **finite sequence** of length n. I will abbreviate the set of all finite sequences of length n with ranges included in X —i.e. $\{0, ...n\} \to X$— by X^n The set of all finite sequences with ranges included in X —i.e. the union of all X^n for $n \ge 0$— is written X^*. The unique member of X^0 is written as _ , and the unique member of $\{x\}^1$ as $\langle x\rangle$. The **catenation** of finite sequences T and U is written as $T.U$, and a catenation $\langle a\rangle.\langle b\rangle.\langle c\rangle$ is written $\langle a, b, c\rangle$.

Since transfinite sequences will not be used in this paper, I'll just say "**sequence**" when "finite sequence" would be more proper.

The **length** of a sequence T is written as $\#T$; for example $\#\langle a, b, c\rangle = 3$. The sequence of length $j - i$ that maps its argument to i plus the argument is written $\langle i, ...j\rangle$; for example $\langle 13, ...16\rangle = \langle 13, 14, 15\rangle$. A sequence **composed** with a sequence gives a sequence of results like this $T \cdot \langle x, y, z\rangle = \langle T(x), T(y), T(z)\rangle$. An identity using all three of these notations is $T \cdot \langle 0, ...\#T\rangle = T$

Two slight abuses of notation will be tolerated when dealing with sequences. First, if it is clear that x is not a sequence, then it is ok to write x rather than $\langle x\rangle$ when it is clear that a sequence is required; e.g. $s.1.t$ should be understood to mean $s.\langle 1\rangle.t$. Second, the application of a sequence to an argument may be written as T_i rather than $T(i)$.

1.2.2 Variable Binding Constructs

Quantified formulæ will be written as, for example, $(\exists! x \in X : R : P)$ with X being a set, R being a formula that further restricts x and P being a formula. More than one variable can be bound, for example $(\forall x \in X; y \in Y : R : P)$. **Quantified terms** follow the same pattern except P is replaced by a term. For example $(\Sigma x \in X : R : t)$.

Three abbreviations are used in quantified formulæ and terms, and in comprehensions.

- The condition R may be omitted in which case it defaults to $true$. For example: $(\Sigma x \in X :: f(x))$.
- The set X may be omitted (together with the $\in$) in which case it defaults to the largest set that makes R and P (or t) well defined. For example, if the domain of $even$ is the integers, the term $(\Sigma x : even(x) : 1/x)$ abbreviates
  $$(\Sigma x \in int - \{0\} : even(x) : 1/x) \quad .$$
  This abbreviation should be used with care.
- Finally, t may be omitted (together with the preceding colon), in which case the default is x. For example, $(\min\ i \in Primes : i > j)$ is minimum prime greater than n. This abbreviation only makes sense if a single variable is being bound.

At times it is easiest to describe an object by stating a property of it that uniquely identifies it. For example, the positive square root of two is simply described as "that positive number which when squared yields 2.0". Using common mathematical notation, it is easy, using set comprehension notation, to describe the singleton set containing only the positive square root of 2 as
$$\{x \in \Re : x \geq 0 \wedge x^2 = 2\} \quad .$$
But, this is not what is wanted; we want the sole member of this set. We will use the notation $(x \in X : R)$ to mean the sole member of the set $\{x \in X : R\}$. Thus we can write the positive square root of 2 as
$$(x \in \Re : x \geq 0 \wedge x^2 = 2) \quad .$$
If the set expression $\{x \in X : R\}$ is undefined or is defined, but not a singleton, then the expression $(x \in X : R)$ is undefined. We call this notation **solution comprehension**.

Set comprehension notation can be generalized to be analogous to quantified terms. $\{x \in X : R : t\}$ will be the set of all things obtained by replacing variable x in term t with a value from X such that R is satisfied. We can define such three-part set comprehensions in terms of two-part comprehensions
$$\{x \in X : R : t\} = \{y : (\exists x \in X : R : y = t)\}$$
(where y is not free in R or t). Solution comprehensions are generalized the same way. So for example $(i \in Indecies : SIN(i) = 120884908 : lname(i))$ will be, if defined, the common last name of all people with 120884908 as SIN. In both solution and set comprehensions, it makes sense to allow more than one variable to be bound.

A **lambda expression** is written as $\langle x \in X : R : t \rangle$ and denotes that function f with domain $\{x \in X : R\}$, such that $f(c)$ equals t^x_c —i.e. t with x replaced by c— for all c in the domain. For example, the function that maps even integers to their successors might be written $\langle i \in int : even(i) : i + 1 \rangle$. Since we are modeling finite sequences with functions, lambda expressions make a concise way to write sequences that obey some rule, for example $\langle 0, 1, 4, 9, 16, 25, 36 \rangle$ can be written as $\langle i \in \{0, ...7\} :: i^2 \rangle$. In this paper that is the only use that will be made of lambda expressions, and we will call this special case **sequence comprehension**.

1.2.3 Cartesian Products

Given a sequence of sets S, we write the Cartesian product of S as ΠS or equivalently as

$$S_0 \times S_1 \times \ldots \times S_{\#S-1} \quad .$$

The members of this set are written $(a, b, \ldots, c)$. Note that for any set A, we have $\Pi\langle A \rangle = A$, and that $\Pi\_ = \{(\ )\}$ — $(\ )$ being the empty tuple. I will sometimes write $a/b/\ldots/c$ rather than $(a, b, \ldots, c)$, especially when dealing with "event-response" pairs (defined later).

1.2.4 Tables

A one-dimensional **normal function table** consists of a list of formulæ and a list of terms (of equal length). A normal function table is a term. In any state (assignment of values to the free variables), the table is undefined unless exactly one of the formulae is true. Otherwise the value of the table is the value of the corresponding member of the term list. For example the table

$a = b$	0
$a < b$	$-a$
$a > b$	$+a$

equals 0, $-a$, or $+a$ depending on whether a is equal, less than, or greater than b.

A different sort of table is the **vector equality table**. An example vector equality table is

	$a = b$	$a < b$	$a > b$
$s =$	u	v	w
$t =$	x	y	z

which is interpreted as the formula

$$(a = b \wedge s = u \wedge t = x) \vee (a < b \wedge s = v \wedge t = y) \vee (a > b \wedge s = w \wedge t = z) \quad .$$

Any terms may appear in the row header. The formulae that appear in the column header must be such that exactly one is true in any state.

1.2.5 Try-else

I define "try x else y" to equal x when x is defined and to equal y when x is undefined. For example: "try $5 \div 0$ else 9" equals 9.

1.2.6 Discussion

Modeling sequences as functions is a matter of notational and conceptual economy. It has the disadvantage of requiring a higher order logic in order to quantify over sequences. One could equally well introduce sequences as first-order objects as is done in [Parnas 1994].

Having a uniform syntax for quantifiers and comprehensions is again a matter of notational economy. It is tempting to introduce some semantic uniformity as well, noting that many quantifiers (e.g. universal, existential, summation, maximum, etc.) are related to binary operators (e.g. $\wedge$, $\vee$, $+$, max) in analogous ways. But some of the quantifiers (exists unique, for example) do not fit this patterns.

Of the abbreviated forms of variable binding, the omission of the set X is the only one that can cause trouble. As mentioned, it should be used with discretion. The abbreviation only makes sense when a unique set is determined. For example $\{x : x \notin x\}$ does not make sense, unless one is assuming some closed universe.

The 3-part set comprehension notation can be found in the Z language [Spivey 1987] and even in at least one discrete math textbook [Gries and Schneider 1993], it is closely related to the axiom of replacement in Zermelo-Fraenkel set theory. The notation is only of marginal usefulness in trace assertion specifications, but it provides a nice stepping stone to 3-part solution comprehensions, which are of great usefulness in specifications.

The solution comprehension is rarely used in mathematical writing. It has been independently rediscovered a number of times, with subtle differences in the treatment of the undefined cases. Russell and Whitehead may have been the first [Russell and Whitehead 1910]. In the Z language it is called "definite description" [Spivey 1987]. It is also very similar to the "let" and "solutions" constructs introduced in [Norvell and Hehner 1993], to the demonic choice quantification of [Ward 1994], and to the epsilon function used by Hilbert [Bernays 1935]. When X is a singleton and R is $true$, it is equivalent to the "let" construct found in many functional languages. The recommended English reading of $(x \in X : R : t)$ is "let x in X such that R in t" or "t where x in X is such that R and for $(x \in X : R)$, "that x in X such that R".

Normal function tables are introduced in [Parnas 1992]. Vector equality tables are inspired by table types in the same report, but differ in detail from all of them.

The try-else construct is similar to that introduced in [Norvell and Hehner 1993].

CHAPTER 2 Theory of Leaf Modules

2.1 Introduction

Shortly I will present a mathematical theory of trace specifications entirely divorced from its application to module specifications. However to aid the reader, I will first mention how the theory will be used. A trace specification —as will shortly be defined— consists of six mathematical objects.

- An alphabet ER of **event-response pairs**. Each member of ER is a pair E/R where E is a call to a module (the name of an access program together with the values of all value parameters and global variables that might be read by the module) and R is a possible response of the module. By "possible" what is meant is syntactically possible. For example, if an access program inc takes a single integer value-result parameter, then for all integers i and j, the pair $(inc(i))/j$ should be in ER. The domain of ER will be called the set of **events** and the range of ER will be called the set of **outputs** or **responses**.
- Can is a a set of traces. Each member of Can is a representative of a set of traces that all lead the module to states that are indistinguishable from each other by any future experimentation. Can is called the set of **canonical traces**.
- $Init$ is a canonical trace that represents the initial state of the module. It is called the **initial trace.**
- c is a function that indicates when a particular call to the module is a proper usage of the module. It is called the **competence function**.
- o is a function that indicates what responses the module might make to any call. It is called the **output function**.
- e is a function that indicates what canonical trace represents the trace that results from adding an event/response pair to a trace. It is called the **extension function**.

In addition to these undefined terms, we will define a set of **feasible traces** $Feasible$ that consists of all the traces that might arise from the usage of a module, and a **reduction function** r that reduces any feasible trace to a canonical one.

2.2 Basic Definitions

A **pre-trace specification** is a tuple $(ER, Can, Init, c, o, e)$ where ER is a set of pairs, Can is a subset of the set ER^* of sequences of pairs drawn from ER, $Init \in Can$,

$$c \in Can \times dom(ER) \rightarrow Bool,$$
$$o \in Can \times ER \rightarrow Bool,$$

and

$$e \in \{T;E;O : o(T, E/O) : (T, E/O)\} \rightarrow Can.$$

We define a subset of the traces called $Feasible$ and a function $r \in Feasible \rightarrow Can$ together by mutual recursion (on the length of traces):

$$\_ \in Feasible$$
$$r(\_) = Init$$
$$(T.(E/O) \in Feasible) = (T \in Feasible \wedge o(r(T), E/O))$$
$$r(T.(E/O)) = e(r(T), E/O)$$

A **trace specification** is a pre-trace specification that satisfies the following axioms:

(ts0) $(\forall T;E : T \in Can \wedge c(T, E) : (\exists O :: o(T, E/O)))$.

(ts1) $Init$ is feasible and, if T is in the range of e, then T is feasible.

(ts2) If T is canonical and feasible, then $r(T) = T$.

The theorems and definitions to follow assume we have a trace specification.

Define the **expanded output function**: $\tilde{o} \in Feasible \times ER \rightarrow Bool$ by

$$\tilde{o}(T, E/O) = o(r(T), E/O)$$

and the **expanded competence function** $\tilde{c} \in Feasible \times dom(ER) \rightarrow Bool$ by

$$\tilde{c}(T, E) = c(r(T), E).$$

2.3 A Few Theorems About Trace Specifications

Theorem 0: If T is feasible, then, for any event E such that $\tilde{c}(T, E)$, there is an O such that $T.(E, O)$ is feasible.

Proof: Immediate from (ts0) and the definition of feasible.
QED

Theorem 1: If T is in the range of r, then T is feasible.

Proof: Let T be in the range of r. From the definition of r, T is either $Init$ or in the range of e. In either case, it is feasible by (ts1).
QED

Theorem 2: The range of r is $Can \cap Feasible$.

Proof: From the type of r we know its range is contained in Can and from theorem 1 that its range is contained in $Feasible$, so $ran(r) \subseteq Can \cap Feasible$. Let T be in $Can \cap Feasible$. By (ts2) it is in the range of r. Thus $Can \cap Feasible \subseteq ran(r)$.
QED

For feasible T and U, define $T \equiv U$ iff $r(T) = r(U)$. This equivalence relation is called **specification equivalence**.

Theorem 3: If T is feasible and canonical and $o(T, E/O)$, then

$$T.(E/O) \equiv e(T, E/O) \quad .$$

Proof. Let T be feasible and canonical, and E and O be such that $o(T, E/O)$. By (ts1) we have $o(r(T), E/O)$, thus by the definition of feasible, we have that $T.(E/O)$ is feasible and hence in the domain of r.

$r(T.(E/O))$

$=$ {defn r}

$e(r(T), E/O)$

$=$ {(ts2)}

$e(T, E/O)$

$=$ {By (ts1) this expression is feasible. Apply (ts2).}

$r(e(T, E/O))$

QED

Theorem 4: $T \equiv U$ implies $\tilde{o}(T, E/O) = \tilde{o}(U, E/O)$ and $\tilde{c}(T, E/O) = \tilde{c}(U, E/O)$.

Proof:

$\tilde{o}(T, E/O)$

$=$ {defn $\tilde{o}$}

$o(r(T), E/O)$

$=$ {$T \equiv U$}

$o(r(U), E/O)$

$=$ {defn $\tilde{o}$}

$\tilde{o}(U, E/O)$

And likewise for $\tilde{c}$.

QED

Theorem 5: If $T \equiv U$, and $T.V$ and $U.V$ are both feasible, then $T.V \equiv U.V$.

Proof is by induction on the length of V. The base case is trivial.

Inductive Case: We can assume that $V = W.(E/O)$ (for some W, E and O). Since W is shorter than V (and noting that $T.W$ and $U.W$ must both be feasible) we have, by the induction hypothesis, that $T.W \equiv U.W$.

$r(T.V)$

$=$ {$V = W.(E/O)$}

$r(T.W.(E/O))$

$=$ {defn of r}

$e(r(T.\mathrm{W}), E/O)$

$=$ {$T.W \equiv U.W$}

$e(r(U.W), E/O)$

$=$ {retracing our steps}

$r(U.V)$

QED

The **extensions** of a feasible trace T is the set $\{V : T.V \in Feasible\}$. Two feasible traces are said to be **observationally equivalent** (written $T \cong U$) iff they have the same extensions.

Theorem 6: Specification equivalence is a refinement of observational equivalence. That is

$$T \equiv U \Rightarrow T \cong U \quad ,$$

for all feasible T and U.

Proof: Let T and U be feasible traces such that $T \equiv U$. Let V be a member of the set of extensions of T. We need only prove (by induction on the length of V) that V is also an extension of U.

Base Case: Suppose that $V = \_$.

$$
\begin{array}{ll}
 & U.V \text{ is feasible} \\
= & \{V = \_\} \\
 & U \text{ is feasible} \\
= & \{\text{by assumption}\} \\
 & \text{true}
\end{array}
$$

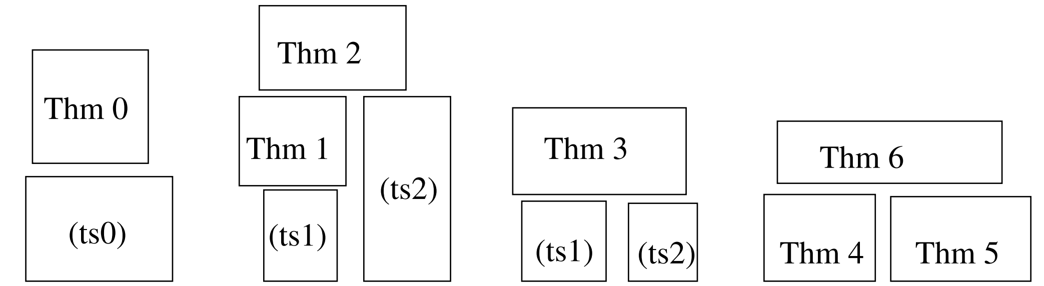
Inductive Case: We can assume that $V = W.(E/O)$ (for some W, E and O). Since $T.V$ is feasible, T.W is feasible and as an induction hypothesis W is an extension of U so we have $U.W$ is also feasible.

$$
\begin{array}{ll}
 & \text{true} \\
= & \{\text{by assumption}\} \\
 & T.V \text{ is feasible} \\
= & \{V = W.(E/O)\} \\
 & T.W.(E/O) \text{ is feasible} \\
= & \{\text{defn of feasible}\} \\
 & T.W \in Feasible \wedge o(T.W, E/O) \\
= & \{T.W \text{ is feasible}\} \\
 & o(r(T.W), E/O) \\
= & \{\text{defn } \tilde{o}\} \\
 & \tilde{o}(T.W, E/O) \\
= & \{\text{theorem 5 } (T.W \equiv U.W) \text{ and theorem 4}\} \\
 & \tilde{o}(U.W, E/O) \\
= & \{\text{retracing our steps}\} \\
 & U.V \text{ is feasible}
\end{array}
$$

QED

2.3.1 Support for the theorems

The following diagram shows how these theorems are supported by the axioms.



The interesting thing is that Theorems 4, 5, and 6 are true of pre-trace specifications.

2.4 Determinism

It often happens that for feasible trace T and event E there is a unique O such that $\tilde{o}(T, E/O)$. In this case, we will generally take the liberty of writing any feasible trace $T.(E/O).U$ as $T.E.U$, since the output component is redundant.

In presentations of trace specifications, we will leave out redundant output components even in the definitions of Can, c, e, and o.

2.5 Discussion

The intended use of trace specifications is to *specify* or to *describe* the behaviour of leaf modules, that is modules that call no modules themselves. The events are just calls to the module's exported routines (access programs in the terminology of [Parnas and Wang 1989] and [Iglewski, Madey, and Stencel 1994]) and the outputs are the results returned from the exported routines. In the case of a *specification*, the set of feasible traces, derived from a trace specification, documents the allowed behaviour of the module. I say *allowed* behaviour because the actual behaviour may be less nondeterministic than the documented behaviour. The expanded competence function specifies those calls which are valid uses of the module. In the case of a *description*, the set of feasible traces documents the *possible* behaviour of the module.

The presentation of trace theory above is based very much on [Iglewski, Madey, and Stencel 1994]. I have used a characteristic function rather than a relation for outputs in order to make it easier to define them simultaneously with the extension function (discussed below). The competence function is new as is $Init$.

The axioms and theorems are new. Axiom (ts0) is an axiom of excluded miracles. Axioms (ts1) and (ts2) are primarily intended to ensure a correspondence between external observations and the abstract states of the module. Without this, the trace specification method devolves into an artificially constrained form of model-based specification. For example, for any trace specification, there is a pre-trace specification that uses in place of each canonical trace its reverse. This pre-trace specification violates (ts2). Axiom (ts2) is an axiom of *realism*.

Axioms (ts1) and (ts2) are a little ugly as they involve derived concepts (feasibility and the reduction function r). However, in practice, I don't think they will be hard to check or hard to meet. These axioms can probably be simplified in the cases where $Init = \_$.

The most important theorem is theorem 6; it states the soundness of the method. Theorem 6 is proved without using the axioms at all. Thus the purpose of the axioms is to ensure realism and miracle exclusion, but not soundness.

I have not insisted that specification equivalence and observational equivalence be equal. This is because making the set of canonical traces small enough that there is a unique canonical trace in every observational equivalence class can make definitions unnatural. Consider —as we will in Chapter 3— a generic set module. The usual way of making the set of canonical traces so small that it contains exactly one representative of each observational equivalence class is to admit only traces sorted by some total order as canonical. But the use of an order to describe a set module seems to me to be artificial. The disadvantage of the approach taken here is that correct implementations may not by provable using the abstraction function technique. This problem can be avoided by using an abstraction relation which relates each state to each canonical trace that it might represent.

None of the sets of events, outputs, or canonical traces are required to be finite. If they are finite, that's ok, but the specification method works equally well for modules with an infinite number of states.

Another reasonable restriction that I have not made would be to insist that all canonical traces be feasible. This would simplify the theory slightly and likely not inconvenience anyone. On the other hand, allowing canonical traces that are not feasible seems only to have the effect of forcing one to pedantically state that a trace is not only canonical but also feasible. Annoying, but not harmful.

The competence function does not restrict the range of the output, so the specifier must still specify what outputs are valid for an invalid event. The occurrence of an event such that the competence function is false is called 'incompetent' use of the module. Various interpretations are possible for what the module should "do" when used incompetently:

- In the **LD interpretation** the module may either loop forever or return from the call, but, if the call is returned from, the output must be as specified by the expanded output function.
- In the **VDM interpretation**, anything at all may happen.

- In the **firewall interpretation**, it is incumbent on the module to “notice” the problem and to cause some remedial action to take place. The remedial action might consist of printing an apologetic message and shutting down the system, calling on some emergency code to restore consistency, or something else that is appropriate to the application. In any case, the error should not go unreported.

Regardless of the interpretation taken, it is the responsibility of the programmers of all clients of a module to ensure that events that make the competence function false do not happen (unless the client module itself is used incompetently).

CHAPTER 3 Presentation of Leaf Module Specifications

3.1 Trace specification documents

The purpose of a trace specification document is to present a trace specification and to connect the events and outputs defined in the trace specification with the actual code of a system. It is important not only that the presentation of the trace specification be complete and unambiguous, but also that it be as easy to read as possible.

I will divide trace specification documents into the following sections:

- Introduction: The name of the module and a description of module parameters.
- Syntax: Definition of ER.
- Dictionary: Definition of auxiliary functions, predicates, and macros.
- Canonical Traces: Definition of Can.
- Initialization: Definition of $Init$.
- Behaviour: Definition of c, o and e.

3.1.1 An Example

Module Set[A : type]

The module is called “Set” and is generic over all types A.

Syntax:

insert	val A	
delete	val A	
in	val A	res Bool

The module has three exported routines. The first parameter of each is a value parameter of type A and the second parameter of routine “in” is a result parameter of type bool.

Each exported routine gives rise to a set called an **event class**. The three event classes here are $\{\,(a \in A)\ ::\ insert(a)\,\}$, $\{\,(a \in A)\ ::\ delete(a)\,\}$, and $\{\,(a \in A)\ ::\ in(a)\,\}$. (Note that these are three-part set comprehensions.) Each routine name acts as a function from the space of its value parameters to some set (it doesn’t matter what set, as long as it's big enough). We assume that each of these functions is one-one (injective) and that their ranges are (pairwise) disjoint. Let us call the event class associated with a routine x, EC_x.

Each exported routine also gives rise to a set called a **response class**. The response classes are formed by tupling the result parameters. In this case, the three response classes are $\{(\ )\}$, $\{(\ )\}$, and $Bool$. Let us call the response class associated with a routine x, RC_x.

Let X be the set of all exported routines, the alphabet of our trace specification is simply

$$ER = (\cup x \in X :: EC_x \times RC_x) \quad .$$

There is no problem with value-result parameters. They simply contribute to both the event class and the response class. It is a good convention to list value parameters first, then value-result parameters, and finally result parameters.

Dictionary: None

In this case, there are no definitions to be made. Definitions can be of predicates or functions, or they can be simply be "text macros".

The dictionary is placed here because it will likely make use of the event and output sets and because the functions defined here may be of use in any of the subsequent sections — declaration before use.

Canonical Trace:

$$Can = \{x \in A^* : (\forall i;j : x_i = x_j : i = j) : \langle i \in \{0, \ldots \#x\} :: insert(x_i) \rangle\}$$

I have decided to use a direct definition of the set. One could instead give the characteristic formula of the set. E.g.

$$(T \in Can) = (\exists x \in A^{\#T} : (\forall i;j : x_i = x_j : i = j) : (\forall i :: T_i = insert(x_i)))$$

Either way seems reasonable.

Initialization: $Init = \_$

This is the common case. The exception is when _ is not canonical. Although this case is common, it is also short, so I suggest it not be made a default.

Behaviour:

The behaviour section presents the routines in the order given in the syntax section. For each routine we must define c, o, and e for each event class.

$e(T, insert(x)) =$

$\exists i :: T_i = insert(x)$	T
$\neg(\exists i :: T_i = insert(x))$	$T.insert(x)$

For “insert” I did not define o. The reason is that “insert” events are associated with exactly one output, namely $(\ )$. I also did not define c for this event class. We will adopt the convention that if c is not mentioned for a particular event class then it is by default defined to be true for all canonical traces and events in the class.

The eagle eyed will have noticed that second argument of e has the wrong type. Strictly, the left hand side of the equation should begin with $e(T,\ (insert(x))/O)$. We will use the convention that if the response part of the second argument to e is not used, it will be omitted.

$$e(T, delete(x)) \ =$$

$\exists!\, i :: T_i = insert(x)$	$(i : T_i = insert(x)\ :\ (T \cdot \langle 0, \ldots i\rangle) . (T \cdot \langle i+1, \ldots \#T\rangle))$
$\neg(\exists!\, i :: T_i = insert(x))$	T

The use of $\exists!$ makes it clear that the solution comprehension is well defined. Looking at the definition of “canonical”, it is clear that $\exists$ could also have been used.

$$o(T, in(x)/b) \ = \ (b = (\exists i :: T_i = insert(x)))$$

$$e(T, in(x)) \ = \ T$$

The definitions for “in” show a nontrivial response class. Thus we give a formula defining o for this event class, but as the response is not used in the definition of e, we omit it.

End of Module Set

3.1.2 Pattern Matching

In [Wang 1994] and [Parnas and Wang 1989] much use is made of pattern matching. Rather than directly supporting pattern matching with any new notations, one can adopt a pattern matching style. The Set module just presented could also be presented in pattern matching style as follows.

In the Dictionary section we define a macro:

$$Match \text{ is } T \ = \ U.insert(x).V$$

(By a “macro” I mean that wherever we see the word Match in the specification, we should mentally replace it with the left hand side. In particular this means that the free variables (T, U, V, and x) receive their bindings at that time. Conversely we should not reckon the free variables of any formula without first performing macro expansion.)

In the Behaviour section we define.

$$e(T, insert(x)) \ =$$

$\exists!\, U;V :: Match$	T
$\neg(\exists!\, U;V :: Match)$	$T.insert(x)$

$$e\,(T, delete\,(x)\,) \;=$$

$\exists!\, U;V :: Match$	$(U;V : Match : U.V)$
$\neg\,(\exists!\, U;V :: Match)$	T

$$o\,(T, in\,(x)\,/\,b) \;=\; (b = \;(\exists!\, U;V :: Match)\,)$$

$$e\,(T, in\,(x)\,) \;=\; T$$

Again the definition of the extension function for “delete” uses a solution comprehension.

3.1.3 Using Vector Equality Tables

In this trace specification, and in others, quite a bit of repetition is caused by the fact that similar tables are used in various formulae. In the present example, we could write the entire contents of the Behaviour section as a single not-too-crowded vector equality table.

	$\exists!\, U;V :: Match$	$\neg\,(\exists!\, U;V :: Match)$
$e\,(T, insert\,(x)\,) \;=$	T	$T.insert\,(x)$
$e\,(T, delete\,(x)\,) \;=$	$(U;V : Match : U.V)$	T
$o\,(T, in\,(x)\,/\,b) \;=$	b	$\neg b$
$e\,(T, in\,(x)\,) \;=$	T	T

3.1.4 Discussion

Even more conciseness can be obtained by using the “try-else” construct. We could write:

$$\begin{pmatrix} e\,(T, insert\,(x)\,)\,, \\ e\,(T, delete\,(x)\,)\,, \\ o\,(T, in\,(x)\,/\,b)\,, \\ e\,(T, in\,(x)\,) \end{pmatrix} = \text{try} \left(U;V : Match : \begin{pmatrix} T, \\ U.V, \\ b = true, \\ T \end{pmatrix} \right) \text{else} \begin{pmatrix} T.insert\,(x)\,, \\ T, \\ b = false, \\ T \end{pmatrix}$$

The three separate bindings and pattern matches of variables U and V have been reduced to one. Using a tabular notation for such expressions could make them more palatable.

The presentation of the trace specification is enhanced by a number of conventions. The exported routines are described in the Behaviour section in the order they appear in the syntax section. It is best to choose this order so as to group similar programs together. The c function is omitted when its value is always true. The o function is omitted when its value is the empty tuple. The e function is present for every exported routine. The third parameter is omitted when it is irrelevant, this allows one to quickly see which routines really depend on the third parameter.

The linkage between the trace specification and the actual code is an issue that I have chosen not to address. For example, in C, should the result of the “in” routine be passed via a pointer parameter, by assignment to a global variable, or using C’s “return” statement?

This sort of question should be answered somewhere —perhaps in the trace specification document, or perhaps in a separate document. In any case such linkage matters should not be allowed confuse the description of trace specifications as mathematical objects.

Another linkage question is how the module is initialized. Again, while this issue may be addressed in the trace specification document, it should not be dealt with by the mathematical trace specification.

3.2 Nondeterminism

In this section we look at two examples with nondeterminism. The first uses traces where we can forget about the fact that the members are really pairs. The second does not.

3.2.1 Independent Nondeterminism

Our first example is another "set" example. We just replace the "delete" and "in" routines with a single routine "get" that provides and deletes an arbitrary member of the set.

Module GetSet[A : type]

Syntax:

insert	val A	
get	res A	res Bool

Dictionary:

$Match$ is $T = U.insert(x).V$

Canonical Trace:

$$Can = \{x \in A^* : (\forall i;j : x_i = x_j : i = j) : \langle i \in \{0, ...\#x\} :: insert(x_i)\rangle\}$$

Initialization: $Init = \_$

Behaviour:

	$\exists! U;V :: Match$	$\neg(\exists! U;V :: Match)$
$e(T, insert(x)) =$	T	$T.insert(x)$

	$T \neq \_$	$T = \_$
$o(T, get(\ )/(x, b)) =$	$b \wedge (\exists U;V :: Match)$	$\neg b$
$e(T, get(\ )/(x, b)) =$	$(U;V : Match : U.V)$	T

In this trace specification, one can see how the output function for "get" describes the acceptable outputs and the extension function for "get" determines the next state based on the output.

I call this sort of nondeterminism **independent** because the selection of outputs is independent of the outputs along the canonical trace.

3.2.2 Dependent Nondeterminism

Our second example is a sort of name server. I have used such a module in a compiler to convert identifiers and keywords into unique numbers suitable for use as array indices. This example also shows a nontrivial c function.

Module IdTable [A : type, N : nats]

Syntax:

enter	val A	res $\{0, \ldots N\}$	res Bool
get	val $\{0, \ldots N\}$	res A	

Dictionary:

$Match$ is $T_i = enter(a) / (x, true)$

Canonical Trace:

$$
\begin{aligned}
Can = \{ & n \in \{0, \ldots N\} ; a \in A^n ; x \in \{0, \ldots N\}^n : \\
& (\forall i;j : a_i = a_j \vee x_i = x_j : i = j) : \\
& \langle i \in \{0, \ldots n\} :: enter(a_i) / (x_i, true) \rangle \}
\end{aligned}
$$

Initialization: $Init = \_$

Behaviour:

	$\exists i;x :: Match$	$\neg(\exists i;x :: Match)$	
		$\#T = N$	$\#T \neq N$
$o(T, enter(a) / (x, b)) =$	$b \wedge (\exists i :: Match)$	$\neg b$	$b \wedge \neg(\exists i;a :: Match)$
$e(T, enter(a) / (x, b)) =$	T	T	$T.(enter(a) / (x, b))$

	$\exists i;a :: Match$	$\neg(\exists i;a :: Match)$
$c(T, get(x) / a) =$	$true$	$false$
$o(T, get(x) / a) =$	$\exists i :: Match$	$true$
$e(T, get(x)) =$	T	T

In the table for “enter”, I have used the abbreviation of stacking up conjuncts, so the headings for the last two columns are really

$$\neg(\exists i;x :: Match) \wedge \#T = N$$

and

$$\neg(\exists i;x :: Match) \wedge \#T \neq N$$

respectively.

The competence function for “get” says that it is incompetent to try to “get” the value associated with an invalid index, i.e. an index that was not produced by the module itself. The output function says that, if the “get” routine is misused, the result could be anything. But the value of the output function for incompetent uses is only meaningful, if the LD interpretation is used.

This example shows what happens when the output is dependent not only on the history of events in the canonical trace, but also on the history of outputs.

3.3 Conclusions

3.3.1 Conciseness and Clarity.

Conciseness and clarity both aid readability and understandability. But there is a tension between conciseness and clarity. Conventions that allow important information to be conveyed without any print are examples: precedence rules that allow parentheses to be omitted, multiplication and composition being written as juxtaposition, the implicit universal quantification of the free variables of a formula are examples.

A fair degree of concision has been obtained. Has clarity been sacrificed? This is up to the reader, but I will point out that the main sources of conciseness are: (a) the use of vector equality table, which allows normally repeated terms to be written only once, (b) the use of macros to allow common subexpressions to be hoisted out of tables, (c) the implicit universal quantification of the variables, (d) the implicit typing of variables, (e) ignoring outputs in traces and definitions when they are irrelevant, and (f) omitting definitions of o and c for some event classes. Let’s look at each of these in turn.

Vector equality tables provide useful structuring and contribute to conciseness at the expense only of repetition. So they add to clarity.

Macros can contribute to page flipping (nonlocallity of relevant information), but it seems to me that they also provide names for useful concepts.

Implicit universal quantification is more of a problem. It makes it impossible to be sure that a misspelling is truly a misspelling rather than a variable with a similar name. It also means that the reader must understand which variables in a defining formula are being defined (c, e and o in our examples), which are already defined (macro names and exported routine names, in our examples), and which are the universally quantified variables (a, b, x, and T, in our examples). An alternative is to allow only the variables mentions in the headers of c, e and o to be implicitly universally quantified.

The implicit typing of variables is another potential problem. I have simply said that the type of such a variable is the largest set such that the expressions in their scope well defined. This is somewhat vague and possibly ambiguous. Extensions to the Hindley-Milner type system such as [Wadler and Blott 1989] may point the way to making this idea precise. In [Parnas 1993], another route is taken, the type of all variables is a universal set and there are special rules for dealing with the undefined values which arise as a result. Unfortunately, these rules are not standard mathematics. Yet another approach is used in the PVS and Larch specification languages [Shankar et al 1993, Garland and Guttag 1990]. In these languages, one can declare that all variables of a certain name have a certain type wherever they are bound. For example one could declare

$$Var\ T \in Can$$

once. This solution is reminiscent of Fortran's IMPLICIT declaration. This last solution is particularly appealing because it combines well with implicit universal quantification: Any variable with an implicit type may be implicitly universally quantified

Ignoring outputs in traces and in applications of e when they are not needed seems to me to only add to clarity. By not mentioning outputs in the definition of canonical traces, one makes clear that the module does not have dependent nondeterminism. By not mentioning outputs in a definition of e, one makes it clear that the result does not depend on the output of the routine (i.e. that the event class has no independent nondeterminism).

Omitting definitions of o is no problem as there is only one possible definition in these cases. Allowing the omission of definitions of c may not be a good idea as it is impossible to distinguish deliberate omissions from oversights.

3.3.2 Comparison with [Wang 1994] and [Parnas and Wang 1989]

The format used in [Wang 1994] and [Parnas and Wang 1989] for presenting trace specifications differs from that presented here in a few respects.

In [Wang 1994] and [Parnas and Wang 1989] the definitions of o and e are in separate sections (there is no equivalent of c) with e coming first. I feel the definition of o on a particular event class is more tightly connected to the definition of e on the same event class, than the definition of o on a different event class (and similarly with o and e switched). I have put these next to each other, together with c for the same class, where they can be easily read together and so they can share the same table, if convenient, as it often is. I put c first since it says for which events o must allow at least one output; next I put o since it defines the domain of e.

In [Wang 1994] and [Parnas and Wang 1989] variables used in tables and in the definitions of canonical traces are implicitly existentially quantified. This is in conflict with the mathematical convention that variables that are free in a formula are implicitly universally quantified. It is easy to find examples where both conventions are used in the same formula. I have chosen to explicitly bind all variables except those that are universally quantified over the whole formula.

In [Wang 1994] and [Parnas and Wang 1989] special kinds of tables are used for pattern matching. The meaning of these tables is less than clear, especially when variables show up in both the pattern column (or conditions column) and the value column. I have endeavoured to make pattern matching nothing special, just ordinary math. The price is that the pattern matching must often be written twice.

The special error tokens used in [Wang 1994] and [Parnas and Wang 1989] have no analogue in the theory of trace specifications that I have used and thus are not present in my presentations of trace specifications. Instead, one can use a result parameter and treat error tokens like any other part of the output.

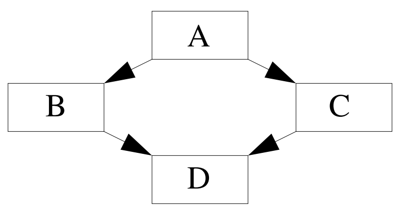
In [Wang 1994] it is not clear how different outputs from the same event can be related and how they in turn are related to the resulting canonical trace. In [Parnas and Wang 1989] the first question is addressed using “output variables”; the second question does not seem to be fully answered. I have tried to make these interactions clearer without using output variables. Indeed output variables were originally intended to describe communication with hardware devices, not to describe nondeterminism. The use of output variable to describe nondeterminism is discussed further in [Iglewski, Mincer-Daskiewicz, Stencel 1994].

CHAPTER 4 Theory and Presentation of Nonleaf Module Specifications

4.1 Introduction

In Chapters 2 and 3 I introduced a new formalization of trace specifications and a new format for presenting trace specifications for use in module specifications.

The trace specification formalism used in those chapters is suitable for specifying leaf modules (modules that do not call other modules apart from submodules). To deal with modules that are not leaves, there are, as I see it, two approaches that can be taken: the **module-as-variable** view and the **module-as-process** view. The best way to see the difference between these is to consider the minimal information needed by a programmer to implement a module using each of these views. Let us suppose that there are modules A, B, C, and D making up a complete program with the following client structure:



That is, A is a client of B and C, and B and C are clients of D. Let us also assume that D contains state which is affected by calls from both B and C and conversely that the behaviour of B and C may be affected by the state of D.

In the module-as-variable view, the programmers implementing B and C are given the specifications for their own modules and that for module D. In the specifications of B and C, module D is treated as a global variable whose value is the canonical trace $r(T)$, for T being the history of D. Global variables are treated by making them both values and results for each exported routine that may access them. The implementor of B (or C) requires also the specification of D in order that they may know how the state of D will be affected by each call they may choose to code. The implementor of B (or C) is free to code any sequence of calls to D that ensures that the final state of D is what is required by the specification of B (or C).

In the module-as-process view, the specifications of B and C say exactly[1] the sequence of calls to be made to D in each case. D is regarded as a complete cipher for the purposes of the specification of B and C. The implementors of B and C need not know the specifica-

1. Modulo nondeterminism.

tion of D and are completely constrained by their own specifications as to how D is to be made use of.

The module-as-variable model has the advantage of generally simplifying the specifications of B and C. However it has the disadvantage that the specifications of B and C depend on the specification of D. If the specification of D were to change, for example, only in the selection of canonical traces, the specifications of B and C would also have to be changed — even though the implementations would not! There may also be cases where module D does not have a trace specification. Consider if module D represents a (human) user of the system. It would be presumptuous to say that two different traces are the same to the user, thus every trace must be its own equivalence class. Another example is where module D represents the file system. Because of concurrent processes that are also using the file system, one can not rely on its state remaining unchanged during the course of the activation of an access program of modules B or C. Again the variable model is not so desirable.

The trace assertion method as outlined in Chapters 2 and 3 is capable of expressing specifications in the module-as-variable view. The state of the called module is considered part of events and part of the output responses. An example will be given in section 4.4.

In the next section we elaborate the trace specification theory to make it suitable for writing specifications in the module-as-process view.

4.2 Running Example

As a running example in this chapter and the next, I will use a “set” module similar to the “GetSet” module described earlier. This module will use another module —presumably a stack— to record additions and deletions to and from the set, so that these can be undone later. The module has the following informal specification (using the process model):

Generic parameters:

- “A” is a type.

Imported routines are:

- Push: takes pair consisting of a token —“add”, “delete”, or “noop”— and an object of type “A”. There is no return value.
- Pop: takes no value parameters, and returns a pair of the sort accepted by “push” and a flag indicating whether the pop was successful.

Exported routines:

- Insert: takes an object “a” of type “A” and adds it to the set. If the object was already in the set, it calls “push” with “noop”, otherwise it calls “push” with “(add, a)”.

- Get: takes no value parameters. If the set is empty, it calls "push" with "noop", and returns "false" as its second result parameter. If the set is nonempty, it returns a member "a" of the set as its first result parameter, "true" as its second result parameter, and as a side effect removes "a" from the set and calls "Push" with "(delete, a)".
- Undo: Takes no value parameters. Calls "pop". If "pop" is unsuccessful, undo returns the token "failed". If the value returned by "pop" is "(add, a)" then "a" is deleted from the set. If the value returned by "pop" is "(delete, a)", then "a" is added to the set. If the value returned by "pop" is "(noop, a)", then no change is made to the set. In all three cases, "undo" returns the token "undone".

Notice that this informal (but rigorous?) specification is using the process model. It makes no assumptions about the specifications of "Push" and "Pop" beyond their syntactic interface. In particular, there is no assumption that they behave in a stack-like manner.

4.3 Two-faced Trace Specification Theory

An **alphabet** A is a set of pairs. A **simple trace** over an alphabet A is a member of A^*. A **trace** over alphabets A and B is a finite sequence of triples $E/S/R$ where $S \in B^*$ and $E/R \in A$. We write $Traces_{A,B}$ for the set of all traces over A and B.

In module specification, one alphabet will be used to represent the calls that can be made on the module (and their syntactically allowed responses). While another alphabet will represent the calls that can be made from the module (and their syntactically allowed responses).

A **pre-two-faced trace specification** is a tuple $(ERT, ERB, Can, Init, c, o, e)$ where ERT and ERB are alphabets such that $rng(ERT) \cap dom(ERB) = \varnothing$. We will use the following abbreviations

ET for $dom(ERT)$
RT for $rng(ERT)$
EB for $dom(ERB)$
RB for $rng(ERB)$.

Continuing: Can is a subset of the set of traces over ERT and ERB, $Init \in Can$,

$c \in Can \times ET \times ERB^* \rightsquigarrow Bool$,
$o \in Can \times ET \times ERB^* \times (RT \cup EB) \rightsquigarrow Bool$,

(where ~> forms the space of partial functions — I will state the required domains of o and c in a short while) and

$e \in \{T;E;S;O \in RT : o(T, E, S, O) : (T, E, S, O)\} \to Can$.

The alphabets ERT and ERB are called the **top alphabet** and the **bottom alphabet**, respectively, and represent, respectively, the interface a module presents to its clients and the union of the interfaces of the modules that serve this module. Traces in Can we call **canonical**. The canonical trace $Init$ is called the **initial trace**. Functions c, o, and e are respectively called the **competence function**, the **output function**, and the **extension function**. The competence function is used to indicate whether a module is being correctly used. The output function is used to indicate the next action the module may take, which can either be a call to a server module or a return. The extension function is used to map behaviours to canonical behaviours so that the output and competence functions need only be defined on a (typically small) subset of all possible histories.

The most important function here is o. It defines what the module might do after each input from the outside. That is if the history of a module is characterized by (T, E, S) — that is the most recent call to the module is E, the history before that time is summarized by the canonical trace T, and the history of interactions with provider modules since the most recent call on the module is S— then $\{X : o(T, E, S, X)\}$ is the set of all responses the module might make. This set may include responses to the caller or calls to provider modules.

We define a subset of the traces called $Feasible$, a subset $Feas$ of $Traces_{ERT, ERB} \times ET \times ERB^* \times (RT \cup EB)$, and a function $r \in Feasible \to Can$ together by mutual recursion (on the length of traces):

$$
\begin{aligned}
&\_ \in Feasible \;\; , \\
&(T.(E/S/O) \in Feasible) = ((T, E, S, O) \in Feas) \;\; , \\
&((T, E, \_, X) \in Feas) = (T \in Feasible \wedge o(r(T), E, \_, X)) \;\; , \\
&((T, E, S'.(E', O'), X) \in Feas) = \\
&\qquad\qquad (\; (T, E, S', E') \in Feas \wedge o(r(T), E, S'.(E', O'), X) \\
&\qquad\qquad\;\; \wedge (X \in RT \Rightarrow (E/X) \in ERT) \;) \;\; , \\
&r(\_) = Init \;\; , \\
&r(T.(E/S/O)) = e(r(T), S, E, O) \;\; .
\end{aligned}
$$

Now we can talk about the domains of c and o. The key idea here is that we need not define these functions for impossible situations, i.e. situations that are provably impossible by looking at the current trace specification only; we make no assumptions about the behaviour of any other modules. The sets $Feas$ and $Feasible$ make this not too difficult. Define a subset D of $Can \times ET \times ERB^*$ by

$$
\begin{aligned}
&((T, E, \_) \in D) = (T \in Feasible) \;\; , \\
&((T, E, S'.(E', O')) \in D) = ((T, E, S', E') \in Feas) \;\; .
\end{aligned}
$$

D is the domain of c and $D \times (RT \cup EB)$ is the domain of o. Two points should be made: First, o is used in the definition of $Feas$ and $Feasible$ which in turn are used to defined the domain of o; the reader should check that this recursion is well founded. Second, the complicated domains affect the presentation of trace specifications only in that they allow the specifier to omit cases that can never arise. Thus the extra complication in the theory makes the specifiers job easier, not harder.

A **two-faced trace specification** is a pre-two-faced trace specification that satisfies the following axioms:

(ts0) $(\forall T;E;S : T \in Can \wedge c(T, E, S) : (\exists X :: o(T, E, S, X)))$.

(ts1) $Init$ is feasible and, if T is in the range of e, then T is feasible.

(ts2) If T is canonical and feasible, then $r(T) = T$.

In the following we assume we have a two-faced trace specification.

Define the **expanded output function**:

$$\tilde{o} \in Feasible \times ET \times ERB^* \times (RT \cup EB) \to Bool$$

by

$$\tilde{o}(T, E, S, O) = o(r(T), E, S, O)$$

and the **expanded competence function** $\tilde{c} \in Feasible \times ET \times ERB^{*} \to Bool$ by

$$\tilde{c}(T, E, S) = c(r(T), E, S) .$$

Note that when the bottom alphabet is empty, there is exactly one simple trace over the bottom alphabet. Thus the set of two-faced trace specifications, such that the bottom alphabet is empty, is isomorphic to the set of trace specifications as defined in chapter 2.

When writing feasible traces, we write $T.(E/S/O).U$ as $T.(E/\ \ /O).U$, when S is completely determined by T and E, and as $T.E.U$, when both S and O are determined by T and E.

4.4 Presentations

We illustrate the variable and the process models by presenting the running example in both styles. The specification is presented in sans-serif type while comments are in roman type.

4.4.1 A Process-Model Specification

Module GetSetWithUndo[A: type]

Imports:

push	val $\{add, delete, noop\} \times A$	
pop	res $\{add, delete, noop\} \times A$	res Bool

Exports:

insert	val A	
get	res A	res Bool
undo	res $\{undone, failed\}$	

Dictionary:

$Match$ is $T = U.insert(x).V$

Canonical Trace:

$$Can = \{x \in A^{*} : (\forall i;j : x_i = x_j : i = j) : \langle i \in \{0, ...\#x\} :: insert(x_i)\rangle\}$$

Initialization: $Init = \_$

Behaviour:

Insert

	$\exists!\, U;V :: Match$	$\neg(\exists!\, U;V :: Match)$
$o(T, insert(x), \_, X)$	$X = push((\text{noop}, x)$	$X = push((\text{add}, x))$
$o(T, insert(x), \langle Y\rangle, X)$	$X = (\ )$	$X = (\ )$
$e(T, insert(x)) =$	T	$T.insert(x)$

Get

	$T \neq \_$	$T = \_$
$o(T, get(\ ), \_, X) =$	$(\exists U;V;x : Match : X{=}push((\text{delete}, x)))$	$(\exists x :: X{=}push((\text{delete}, x)))$
$o(T, get(\ ), \langle Y\rangle, X) =$	$(\exists x :: Y{=}push((\text{delete}, x))$ $\wedge X{=}(x, true))$	$(\exists x :: X{=}(x, false))$
$e(T, get(\ ), S, (x, b)) =$	$(U;V : Match : U.V)$	T

Undo

$o(T, undo(\ ), \_, X) = (X{=}pop(\ ))$

	b	$\neg b$
$o(T, undo(\ ), \langle pop(\ ) / ((c, x), b)\rangle, X) =$	$X = \text{undone}$	$X = \text{failed}$

		$e(T, undo(\ ), \langle pop(\ ) / ((c, x), b)\rangle) =$
$\neg b$		T
b	$c = \text{delete}$	$T.insert(x)$
	$c = \text{add}$	$(U;V : Match : U.V)$
	$c = \text{noop}$	T

4.4.2 A Variable-Model Specification

Here we respecify the same module using the variable model and trace specifications as defined in Chapter 2. The state of the stack module is represented by parameters S and S'. The type of these is $SCan$, the type of all canonical traces for the stack.

In this specification, we assume that the stack is bounded by an integer N>0 and that the stack is really a stack, except that, when it is full, the oldest item is dropped.

Module GetSet [A: type]

Syntax:

insert	val A	val SCan	res SCan	
get	val SCan	res SCan	res A	res Bool
undo	val SCan	res SCan	res $\{\text{undone}, \text{failed}\}$	

Dictionary:

$Match$ is $T = U.insert(x).V$
$Push((c, x))$ is

$\#S = N$	$\#S < N$
$S' = (S \cdot \langle 1, \ldots N\rangle).push((c, x))$	$S' = S.push((c, x))$

Pop is $S' = (S \cdot \langle 0, \ldots \#S - 1\rangle)$
$Top(c, x)$ is $S_{\#S-1} = push((c, x))$
$Undo$ is

$(\exists x :: Top(\mathrm{add}, x))$	$(U;V;x : (Top(\mathrm{add}, x) \wedge Match) : U.V)$
$(\exists x :: Top(\mathrm{delete}, x))$	$(x : (Top(\mathrm{delete}, x)) : T.insert(x))$
$(\exists x :: Top(\mathrm{noop}, x))$	T

Canonical Trace:

$$Can = \{x \in A^* : (\forall i;j : x_i = x_j : i = j) : \langle i \in \{0, \ldots \#x\} :: insert(x_i)\rangle\}$$

Note this is a bit of a cheat as we should really include all the value parameters in the canonical trace. But, as the stack parameter is never used, we will omit mention of it.

Initialization: $Init = \_$

Behaviour:

	$\exists! U;V :: Match$	$\neg(\exists! U;V :: Match)$
$o(T, insert(x, S) / S') =$	$(\exists x :: Push((\mathrm{noop}, x)))$	$Push((\mathrm{add}, x))$
$e(T, insert(x, S)) =$	T	$T.insert(x)$

	$T \neq \_$	$T = \_$
$o(T, get(S) / (S', x, b)) =$	$b \wedge (\exists U;V :: Match) \wedge Push((\mathrm{delete}, x))$	$\neg b \wedge (\exists x :: Push((\mathrm{noop}, x)))$
$e(T, get(S) / (S', x, b)) =$	$U;V : Match : U.V$	T

	$S \neq \_$	$S = \_$
$o(T, undo(S) / (S', f)) =$	$Pop \wedge f = \mathrm{undone}$	$S' = S \wedge f = \mathrm{failed}$
$e(T, undo(S)) =$	$Undo$	T

CHAPTER 5 Automata Theoretic Models

One can think of the trace assertion method as being a concise way of presenting automata with a large number of states and transitions. The main constraint on the automata described is that the states be actual histories of the module and that, if the history of a module happens to be a state, then the state of the automaton is that history. Those histories chosen as states, are what we have been calling canonical traces. This view of trace specifications has been investigated in [Janicki 95] and [Iglewski, Madey, and Stencel 94]. Both these papers and the present chapter deal with deterministic automata only, in the sense that for each state and label, there is only one possible next state. Chapter 6 and [Wang 1994] consider a form of trace specification that would be best modeled with non-deterministic automata.

The Z and VDM methods of module specification may be viewed as having (or being) similar automata theoretic models, but without any restriction as to the nature of the set of states used.

5.1 Dichromatic Automata

To keep things simple, we begin by considering a class of automata for modeling trace specifications as defined in Chapter 2. This case is similar to those considered in the references, but looking ahead to the case where the bottom alphabet is nonempty causes us to use a more elaborate kind of automata than one might at first think of.

A **dichromatic automaton** is a tuple (ER, B, G, i, bg, gb) where ER is an alphabet (we abbreviate $dom(ER)$ by ET and $rng(ER)$ by RT), B and G are disjoint sets, $i \in B$,

$$bg \in B \times ET \sim> G \quad ,$$

$$gb \in G \times RT \sim> B \quad ,$$

(where $\sim>$ forms the space of partial functions), and, finally,

$$(\forall b0;g;b1;E;O :: b1 = gb(bg(b0, E), O) \Rightarrow (E/O) \in ER) \quad .$$

We call the set B the set of **black states** and the set G the set of **green states**.

The language L_A accepted by the automaton is a set of traces defined by recursion together with a function $\beta \in L_A \to B$, a set $LG_A \subseteq ER^* \times ET$ and another function $\gamma \in LG_A \to G$

$$\_ \in L_A \quad ,$$

$$(T\,.\,(E/O) \in L_A) = ((T, E) \in LG_A \wedge (\gamma(T, E), O) \in dom(gb)) \quad ,$$

$$\beta(\_) = i \quad ,$$

$$\beta(T\,.\,(E/O)) = gb(\gamma(T, E), O) \quad ,$$

$$((T, E) \in LG_A) = (T \in L_A \wedge (\beta(T), E) \in dom(bg)) \quad ,$$

$$\gamma(T, E) = bg(\beta(T), E) \quad .$$

A trace specification $TS = (ER, Can, Init, c, o, e)$ determines an automaton

$$DA(TS) = (ER, Can, G, init, bg, gb) ,$$

where

$$G = dom(bg) = dom(gb) = \{T;E;O : o(T, E/O) : (T, E)\} ,$$
$$bg(T, E) = (T, E) ,$$
$$gb((T, E), O) = e(T, E/O)) .$$

It is clear that $L_{DA(TS)} = Feasible$. It is also clear that two trace specifications that map to the same automaton differ only in their competence functions.

5.2 Trichromatic Automata

Dichromatic automata are a mere warm up exercise for the objects we are really interested in: **trichromatic automata**, which model two-faced trace specifications. A trichromatic automaton is a tuple

$$(ERT, ERB, B, G, R, i, bg, gb, gr, rg) ,$$

in which ERT and ERB are alphabets (we use the same abbreviations for range and domain as with two-faced trace specifications), B, G, and R are pairwise disjoint sets. $i \in B$,

$$bg \in B \times ET \rightsquigarrow G ,$$
$$gb \in G \times RT \rightsquigarrow B ,$$
$$gr \in G \times EB \rightsquigarrow R ,$$
$$rg \in R \times RB \rightsquigarrow G ,$$

with two additional restrictions to be mentioned later. We call R the set of **red states**.

As before, we can define the language L_A of traces accepted by an automaton A. We define it simultaneously with sets $LG_A \subseteq Traces_{ERB,\, ERT} \times ET \times ERB^*$ and $LR_A \subseteq Traces_{ERB,\, ERT} \times ET \times ERB^* \times EB$, and functions $\beta \in L_A \rightarrow B$, $\gamma \in LG_A \rightarrow G$ and $\rho \in LR_A \rightarrow R$. The definitions are:

$$\_ \in L_A ,$$
$$(T . (E/S/O) \in L_A) = ((T, E, S) \in LG_A \wedge (\gamma(T, E, S), O) \in dom(gb)) ,$$
$$\beta(\_) = i ,$$
$$\beta(T . (E/S/O)) = gb(\gamma(T, E, S), O) ,$$
$$((T, E, \_) \in LG_A) = (T \in L_A \wedge (\beta(T), E) \in dom(bg)) ,$$
$$((T, E, S'. (E', R')) \in LG_A) = ((T, E, S', E') \in LR_A \wedge (\rho(T, E, S', E'), R) \in dom(rg)) ,$$
$$\gamma(T, E, \_) = bg(\beta(T), E) ,$$
$$\gamma(T, E, S'. (E', R')) = rg(\rho(T, E, S', E'), R') ,$$
$$((T, E, S, E') \in LR_A) = ((T, E, S) \in LG_A \wedge (\gamma(T, E, S), E') \in dom(gr)) ,$$
$$\rho(T, E, S, E') = gr(\gamma(T, E, S), E') .$$

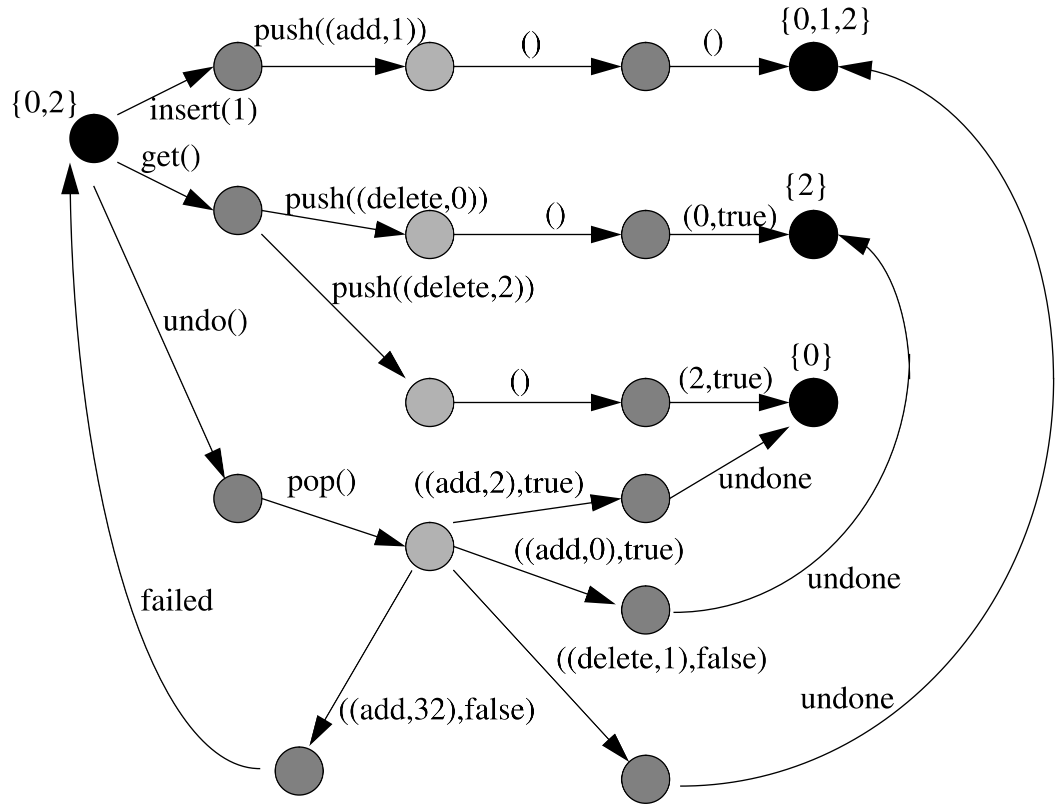
I hope the reader will take the time to see that although this list of definitions looks a bit intimidating, it consists of only a few ideas used repeatedly. The only complication is that there are two ways to get to a green state, from a black state and from a red state. The additional restrictions on automata referred to earlier, are now simply stated as

$$T . (E, S, O) \in L_A \Rightarrow (E/O) \in ERT$$

and

$$((T, E, S'. (E'/R')) \in LG_A) \Rightarrow (E'/R') \in ERB \quad .$$

The following figure illustrates a portion of a trichromatic automaton constructed from our running example.



We can generalize our embedding in a straight-forward way. Given a two-faced trace specification $TS = (ERT, ERB, Can, Init, c, o, e)$, we define a trichromatic automaton

$$TA(TS) = (ERT, ERB, Can, G, R, init, bg, gb, gr, rg) \quad ,$$

in which

$$G = \{T;E;S;X : o(T, E, S, X) : (T, E, S)\} \quad ,$$
$$R = \{T;E;S;E' : o(T, E, S, E') \wedge O \in EB : (T, E, S, E')\} \quad ,$$
$$dom(bg) = \{T;E;X : o(T, E, \_, X) : (T, E)\} \quad ,$$
$$bg(T, E) = (T, E, \_) \quad ,$$
$$dom(gb) = \{T;E;S;O : o(T, E, S, O) \wedge O \in RT : ((T, E, S), O)\} \quad ,$$
$$gb((T, E, S), O) = e(T, E, S, O) \quad ,$$
$$dom(gr) = \{T;E;S;E' : o(T, E, S, E') \wedge E' \in EB : ((T, E, S), E')\} \quad ,$$
$$gr((T, E, S), E') = (T, E, S, E') \quad ,$$

and

$$dom(rg) = \{T;E;S;E';O';X : o(T, E, S. (E'/O'), X) : ((T, E, S, E'), O')\} \; ,$$
$$rg((T, E, S, E'), O') = (T, E, S. (E', O')) \quad .$$

As before we have $L_{TA(TS)} = Feasible$ and that two two-faced trace specifications that map to the same automaton differ only in their competence functions.

Note that this construction does not yield a minimal automaton. In fact, it may even create an infinite state automaton where a finite state automaton would be sufficient.

Conversely, given a trichromatic automaton A we can construct a two-faced trace specification using the quotient construction outlined in [Janicki 1994]. Specifically, let A be a trichromatic automaton $(ERT, ERB, B, G, R, i, bg, gb, gr, rg)$. Define an equivalence relation on L_A

$$(T \cong U) = (\forall V :: (T.V \in L_A) = (U.V \in L_A)) .$$

Let ε be any function at all from the equivalence classes of $\cong$ to L_A, such that $\varepsilon(C) \in C$, for all equivalence classes C (i.e. a choice function). Let

$$Can = \{T \in L_A :: \varepsilon([T]_{\cong})\} \quad ,$$
$$Init = \varepsilon([\_]_{\cong}) \quad ,$$
$$c(T, E, \_) = (\beta(T), E) \in dom(bg) \quad ,$$
$$c(T, E, S.(E', O')) = (\rho(T, E, S, E'), O') \in dom(rg) \quad ,$$
$$o(T, E, S, X) = ((\gamma(T, E, S), X) \in dom(gb) \vee (\gamma(T, E, S), X) \in dom(gr)) \quad ,$$

and

$$e(T, E, S, O) = \varepsilon([T.(E/S/O)]_{\cong}) \quad .$$

Now $Q(A) = (ERB, ERT, Can, Init, c, e, o)$.

CHAPTER 6 Exotic Nondeterminism

6.1 An Example

Trace specification theory as presented in Chapter 2 is able to deal with nondeterminism as demonstrated in the examples of Chapter 3. As shown in Chapter 5, the automata theoretic models, are however still deterministic. Here is an example module that is quite difficult to specify using the methods of the preceding chapters. It is a set module with three exported routines: insert(x) adds x to the set, if it is not already in; remove() removes an arbitrary member, if the set is nonempty; and in(x) reports on whether x is still in the set.[1] The reader should stop here and attempt to specify this module with the methods presented above.

Thinking about this from an automata theoretic point of view, one finds that the canonical traces (black states) used in our earlier “set” modules are the easiest to work with, but that the automaton must be nondeterministic. In terms of trace specifications, the extension function must give not one result, but a choice of results.

This leads us to the following...

6.2 Definitions

An **exotic pre-trace specification** is a tuple $(ER, Can, Init, c, o, e)$ where ER is a set of pairs, Can is a subset of the set ER^* of sequences of pairs drawn from ER, $Init \in Can$,

$$c \in Can \times dom(ER) \to Bool,$$
$$o \in Can \times ER \to Bool,$$

and

$$e \in \{T;E;O;T' : o(T, E/O) : (T, E/O, T')\} \to Bool.$$

The difference is in the extension function, which is now a characteristic predicate for a set of next states.

We define $Feasible \subseteq ER^*$ and $r \in Feasible \to 2^{Can}$ by recursion:

$$\_ \in Feasible \ ,$$
$$r(\_) = \{Init\} \ ,$$
$$(T.(E/O) \in Feasible) = (T \in Feasible \wedge (\exists C \in r(T) :: o(C, E/O))) \ ,$$
$$r(T.(E/O)) = (\cup C \in r(T) : o(C, E/O) : \{T' : e(C, E/O, T')\}) \ .$$

1. This example was told to me by David Parnas, who heard it from Yabo Wang.

An **exotic trace specification** is an exotic pre-trace specification that satisfies the following axioms:

(ts0a) $(\forall T;E : T \in Can \wedge c(T, E) : (\exists O :: o(T, E/O)))$

(ts0b) $(\forall T;E;O : T \in Can \wedge o(T, E/O) : (\exists T' :: e(T, E/O, T')))$

(ts1) $Init$ is feasible and, if T' is such that $(\exists T;E;O :: e(T, E/O, T'))$ then T' is feasible.

(ts2) If T is canonical and feasible, then $T \in r(T)$.

The theorems and definitions to follow assume we have an exotic trace specification.

Define the **expanded output function**: $\tilde{o} \in Feasible \times ER \to Bool$ by

$$\tilde{o}(T, E/O) = (\exists C \in r(T) :: o(C, E/O))$$

and the **expanded competence function** $\tilde{c} \in Feasible \times dom(ER) \to Bool$ by

$$\tilde{c}(T, E) = (\forall C \in r(T) :: c(C, E)) .$$

6.3 Theorems

We can now prove theorems analogous to those in Chapter 2.

Lemma 0: If T is feasible, then $r(T)$ is nonempty.

Proof: by induction on the length of T. The base case is trivial. The induction case:
Let $T.(E/O)$ be feasible, by the definition of feasible we know that T is feasible and that $(\exists C \in r(T) :: o(C, E/O))$. Now we prove

$r(T.(E/O)) \neq \{\ \}$

= {definition of r}

$(\cup C \in r(T) : o(C, E/O) : \{T' : e(C, E/O, T')\}) \neq \{\ \}$

= {set theory}

$(\exists C \in r(T) : o(C, E/O) : \{T' : e(C, E/O, T')\} \neq \{\ \})$

= {set theory}

$(\exists C \in r(T) : o(C, E/O) : (\exists T' :: e(C, E/O, T')))$

= {ts0b}

$(\exists C \in r(T) : o(C, E/O) : true)$

= {trading}

$(\exists C \in r(T) :: o(C, E/O))$

= {from definition of feasible}

$true$

QED

Theorem 0: If T is feasible, then, for any event E such that $\tilde{c}(T, E)$, there is an O such that $T.(E, O)$ is feasible.

Proof: Let T be feasible and E be any event such that $\tilde{c}(T, E)$. By Lemma 0, we know that $r(T)$ is nonempty. Let C be any member of $r(T)$. Now we prove the consequent

$(\exists O :: ((T.(E, O)) \in Feasible))$

= {Definition of feasible}

$(\exists O :: T \in Feasible \wedge (\exists C \in r(T) :: o(C, E/O)))$

$=$ {Assumption that T is feasible}
$(\exists O :: (\exists C \in r(T) :: o(C, E/O)))$
$\Leftarrow$ {Generalization}
$(\exists O :: o(C, E/O))$
$\Leftarrow$ {(ts0a)}
$c(C, E)$
$\Leftarrow$ {Definition of $\tilde{c}$}
$\tilde{c}(T, E)$
$=$ {Assumption}
$true$

QED

Theorem 1: If T is a member of a set in the range of r, then T is feasible.

Proof: By the definition of r, T is either $init$ or such that

$$o(C, E/O) \wedge e(C, E/O, T)$$

for some canonical C. In either case it is feasible by $(ts1)$.
QED

Theorem 2: For any trace T', $(T' \in Can \cap Feasible) = (\exists T :: T' \in r(T))$.

Proof: The $\Leftarrow$ direction follows from the type of r and from Theorem 1.
The $\Rightarrow$ direction follows from (ts2).
QED

For feasible T and U, define $T \equiv U$ iff $r(T) = r(U)$. This equivalence relation is called **specification equivalence**.

Theorem 3: There does not appear to be a direct analogue of Theorem 3 from Chapter 2.

Theorem 4: $T \equiv U$ implies $\tilde{o}(T, E/O) = \tilde{o}(U, E/O)$ and $\tilde{c}(T, E/O) = \tilde{c}(U, E/O)$.

Proof is direct from the definitions.

Theorem 5: If $T \equiv U$, and $T.V$ and $U.V$ are both feasible, then $T.V \equiv U.V$.

Proof is by induction on the length of V. The base case is trivial. For the inductive case assume that $V = W.(E/O)$ (for some W, E, and O). Since W is shorter than V (and noting that $T.W$ and $U.W$ are both feasible) we have by the induction hypothesis that $T.W \equiv U.W$. Now to prove the consequent:

$r(T.V)$
$=$ $\{V = W.(E/O)\}$
$r(T.W.(E/O))$
$=$ {Definition of r}
$(\cup C \in r(T.W) : o(C, E/O) : \{T' : e(C, E/O, T')\})$
$=$ $\{T.W \equiv U.W\}$
$(\cup C \in r(U.W) : o(C, E/O) : \{T' : e(C, E/O, T')\})$
$=$ {Retracing our steps}

$r(U.V)$

QED

The **extensions** of a feasible trace T is the set $\{V : T.V \in \mathit{Feasible}\}$. Two feasible traces are said to be **observationally equivalent** (written $T \cong U$) iff they have the same extensions.

Theorem 6: Specification equivalence is a refinement of observational equivalence. That is

$$T \equiv U \Rightarrow T \cong U$$

for all feasible T and U.

Proof: Let T and U be feasible traces such that $T \equiv U$. Let V be a member of the set of extensions of T. We need only prove (by induction on the length of V) that V is also an extension of U. The base case is easy. For the induction case we assume that $V = W.(E/O)$, for some W, E, and O. Since $T.V$ is feasible, $T.W$ is also feasible. By induction we have that $U.W$ is feasible. From Theorem 5, we have $T.W \equiv U.W$.

$$\begin{aligned}
& \text{true} \\
= \quad & \{\text{By assumption.}\} \\
& T.V \text{ is feasible} \\
= \quad & \{V = W.(E/O)\} \\
& T.W.(E/O) \text{ is feasible} \\
= \quad & \{\text{Definition of feasible}\} \\
& T.W \in \mathit{Feasible} \wedge (\exists C \in r(T.W) :: o(C, E/O)) \\
= \quad & \{\dot{T}W \text{ is feasible}\} \\
& (\exists C \in r(T.W) :: o(C, E/O)) \\
= \quad & \{T.W \equiv U.W \text{ and definition of } \equiv\} \\
& (\exists C \in r(U.W) :: o(C, E/O)) \\
= \quad & \{\text{retracing our steps}\} \\
& U.V \text{ is feasible}
\end{aligned}$$

QED

6.4 Presentation

An example should suffice to show the analogue of the presentation format of Chapter 3. This is a specification of the module described at the start of this Chapter.

Module ExoticSet[A: type]

Syntax:

insert	val A	
delAny		
in	val A	res Bool

Dictionary:

Match is $T = U.\mathit{insert}(x).V$

Canonical Trace:

$$Can = \{x \in A^* : (\forall i;j : x_i = x_j : i = j) : \langle i \in \{0, ...\#x\} :: insert(x_i)\rangle\}$$

Initialization: $Init = \_$

Behaviour:

$e(T, insert(x), T') =$

$\exists U;V :: Match$	$T' = T$
$\neg(\exists U;V :: Match)$	$T' = T.insert(x)$

$e(T, delAny(\ ), T') =$

$T \neq \_$	$\exists U;V;x : Match : T' = U.V$
$T = \_$	$T' = T$

$o(T, in(x) / b) = (b = (\exists U;V :: Match))$

$e(T, in(x), T') = (T' = T)$

End of Module Set

CHAPTER 7 Concluding Remarks

7.1 Further Work

7.1.1 Nonleaf Modules

Chapter 4 presents two approaches to nonleaf modules. Neither seems to me to be entirely satisfactory. Some situations may call for the variable approach, others for the process approach. There may be other situations in which some other approach is called for.

7.1.2 Automata Theory of Exotic Nondeterminism

Nondeterministic di- and trichromatic automata could be defined.

7.1.3 Data refinement

The proof of one trace specification refining another or of a program or automaton implementing a trace specification has not been addressed in this report. The methods of [Hoare, He, and Sanders 1987] should be applicable.

7.1.4 Multiple Objects

Some modules implement multiple similar objects. Consider a module that manages a number of sets of the sort described in our "Set" module. It seems a pity to throw away the clean single object trace specifications in such a case. One proposal is to project the trace of the module onto those calls that are relevant to a particular object. This is fine in the case where the objects have no interaction. In cases where they have some interaction (e.g. competition for limited resources) there is some question as to how to specify them.

7.2 Acknowledgments

I am pleased to thank the members of the Software Engineering Research Group at McMaster and Krzysztof Stencel for many useful discussions, the Communications Research Lab at McMaster for providing office space and a workstation, and NSERC for funding.

CHAPTER 8 Bibliography

[Bernays 1935] Paul Bernays, ‘Hilberts Untersuchungen über die Grundlagen der Arithmetik’, in *Gesammelte Abhandlungen*, by David Hilbert, Springer, 1935.

[Gries and Schneider 1993] David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993.

[Garland and Guttag 1990] S. Garland and J. Guttag. *A guide to LP, The Large Prover, Massachusetts Institute of Technology: Preliminary Draft*, August 1990.

[Hoare, He, and Sanders 1987] C.A.R. Hoare, He Ji Feng, and Jeff Sanders, ‘Prespecification in data refinement’. *Inf. Proc. Lett.*, Vol 25, pp71-76, 1987.

[Iglewski, Madey, and Stencel 1994] M. Iglewski, J. Madey, and K. Stencel, ‘On the Fundamentals of the Trace Assertion Method’, Technical Report RR 94/09-6 Départment D’Informatique, Université du Québec à Hull, September 1994.

[Iglewski, Mincer-Daskiewicz, Stencel 1994] M. Iglewski, J. Mincer-Daskiewicz, and K. Stencel, ‘Some experiences with Specification of Nondeterministic Modules’, Technical Report RR 94/09-7, Départment D’Informatique, Université du Québec à Hull, 1994.

[Janicki 1995] Ryszard Janicki, ‘On Pure and Polluted Traces, Quotient Automata and the “Sink” State’, unpublished draft.

[Norvell and Hehner 1993] Theodore S. Norvell and Eric C. R. Hehner, ‘Logical Specifications for Functional Programs’, in *Mathematics of Program Construction*, Bird, Morgan, and Woodcock (Eds.), LNCS 669, Springer-Verlag, 1993.

[Parnas 1992] David Lorge Parnas, *Tabular Representation of Relations*, CRL Report 260, October 1992.

[Parnas 1993] David Lorge Parnas, ‘Predicate Logic for Software Engineering”, *IEEE Transactions on Software Engineering*, Vol 19, No. 9, September 1993.

[Parnas and Wang 1989] David Lorge Parnas and Yabo Wang, *The Trace Assertion Method of Module Interface Specification,* Queen’s TRIO Technical Report 89-261, 1989.

[Shankar et al 1993] N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb 1993.

[Spivey 1987] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, 1987.

[Wadler and Blott 1989] P. Wadler and S. Blott, ‘How to make *ad hoc* Polymorphism less *ad hoc*’, In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, pp 60-70, January 1989.

[Ward 1994] Nigel Ward, *A Refinement Calculus for Nondeterministic Expressions*, PhD thesis, University of Queensland, 1994.

[Wang 1994] Yabo Wang, *Specifying and Simulating the Externally Observed Behaviour of Modules,* PhD thesis McMaster University, CRL report 292, August 1994.

[Whitehead and Russell 1910] Alfred North Whitehead and Bertrand Russell, ‘Incomplete Symbols: Descriptions’, in *From Frege to Gödel*, Jean van Heijennoort Ed., Harvard, 1967.