

# Developing Black Box Specifications Through Sequence Enumeration

S. J. Prowell

Computer Science Department, The University of Tennessee  
107 Ayres Hall, Knoxville, TN 37996-1301, USA

Q-Labs, Inc.

5516 Lonas Road, Suite 110, Knoxville, TN 37909, USA

Email: sprowell@cs.utk.edu

## Abstract

*A rigorous behavioral specification can greatly reduce risk by exposing ambiguities in requirements and making explicit otherwise tacit information. Such an external, or “black box” specification can be developed from behavioral requirements in a systematic manner through the process of sequence enumeration. This process results in an arguably complete, consistent, and traceable specification of external system behavior. Sequence abstraction provides a powerful means to manage and focus the enumeration process.*

## 1. Motivation

The box structure development method describes a software system through three levels of abstraction [6]. The first view is the *black box* view, which describes only external behavior. The second view is the *state box* view, which is derived from the black box by adding internal state. The third view, the *clear box*, is an algorithmic implementation of the state box. The three views form a hierarchy of abstractions which allows for stepwise refinement and verification as each view is derived from the previous. Each view must be complete, consistent, and its correctness with respect to any previous view must be justified.

The box structure method has been successfully applied to large industry problems (see, for example, [7]) and its relationship to current practices such as object-oriented design has been studied (see, for example, [3]). For a full technical presentation of box structures, see [5].

The black box must be developed from the requirements, which are often incomplete and may even be inconsistent. This paper presents a method for developing a black box specification from requirements such that completeness and consistency are assured, and such that the result is traceable to the requirements. The method considered here, sequence-based software specification, has been used successfully in industrial applications and

provides a systematic, straightforward means to discover a black box specification.

## 2. Black Box Specification

Let  $S$  denote the set of all observable external events for a software system.  $S$  is thus the set of software *stimuli*. Let  $R$  denote the set of all externally-observable software behaviors.  $R$  is thus the set of all software *responses*.  $S^*$  will denote the set of all finite-length sequences of stimuli, where these sequences are taken to represent the historical sequence of stimuli received, read from left to right. The special empty sequence is denoted  $\lambda$ , and sequence concatenation is denoted by juxtaposition, so the concatenation of sequence  $u$  onto the left end of sequence  $v$  is denoted  $uv$ .  $S^+$  will denote the set of all non-empty sequences ( $S^* = S^+ \cup \{\lambda\}$ ).

The black box function for a software system is a total function  $\text{BB}: S^* \rightarrow R$  which maps all sequences of observed stimuli to the appropriate next response. Software in operation need not emit a response after every stimulus in a series, but the mapping rule requires one. Sequences for which there should be no response are mapped to a special value  $0$  called the *null* response, which denotes no software response. For convenience,  $\text{BB}(\lambda) = 0$ .

Some sequences may be physically or logically impossible to observe for software in operation, given the system definition, but the mapping rule requires some value from  $R$  for every sequence. Sequences which are impossible are called *illegal* sequences, and are mapped to the special value  $\omega$ . All decisions about treating sequences as illegal must be justified based on an assessment of risk [10].

The black box specification may be written using some box structure notation, such as the box description language of [6] or [2], or in a tabular notation similar to that used in [8]. Black box specifications are often written using connectives of the form “more recently than,” “since the latest,” and “prior to the most recent” to describe circumstances in the history of stimuli (i.e., to describe

stimulus sequences). Such statements, while precise, are often difficult to examine for completeness and consistency. Further, when a box description language is used, the result may be very close to a process notation, and thus impose assumptions about the final design of the system. To avoid these potential problems, a special notation called *prefix recursion* is introduced.

## 2.1. Writing Prefix-Recursive Functions

A *prefix-recursive function* is any function  $f: X^* \rightarrow Y$  (where  $X$  and  $Y$  are arbitrary finite sets) which is written in the following form:

$$f(\lambda) = y_0$$

$$\forall u \in X^*, \forall x \in X: \begin{cases} f(ux) = y_1 \text{ if } p_1(ux) \\ f(ux) = y_2 \text{ if } p_2(ux) \\ \dots \\ f(ux) = y_n \text{ if } p_n(ux) \\ f(ux) = f(u) \text{ otherwise.} \end{cases}$$

The  $y_0, y_1, \dots, y_n$  are elements of  $Y$  (they need not be constant; one can use a function on  $X^*$  instead, but  $y_0$  must obviously be a constant) and the  $p_1, p_2, \dots, p_n$  are predicates on  $X^*$  called *component predicates*.

Such a representation of functions has advantages for use in black box specifications. First, it is fairly easy to discover and write such specification functions; a systematic means exists to attack even complex specification functions. Second, such specification functions may be quickly converted into a state machine, which eases the transition from black box to state box. Third, such a function is no more complex (in a formal language sense) than its most complex component.

A process for creating a prefix-recursive specification function  $f: S^* \rightarrow Y$  is outlined below. In this description,  $u$  will denote the current stimulus sequence up to but not including the current stimulus, which is denoted  $x$ .

1. Decide what information is to be discovered from the stimulus sequence. Give  $f$  a name descriptive of this information.
2. Define the domain of  $f$ . It must be some product of  $S^*$  (possibly just  $S^*$ ).
3. List the possible values of the specification function (the co-domain  $Y$ ).
4. Decide on the value  $y_0$  for  $f(\lambda)$ . This is the basis case of the specification function. Write the rule  $f(\lambda) = y_0$ .
5. Consider reading an arbitrary stimulus history from earliest stimulus to most recent stimulus. Make a list of the conditions under which  $f$ 's value should be changed. (One can think of the value of a function on sequences

changing as stimuli are accumulated into the stimulus history, much as one thinks of the value of a function changing between two points in the domain of a numerical function.)

6. For each condition, consider the stimuli which could cause the change in value. For each such stimulus  $s$ , write a rule of the form

$$\forall u \in S^*, \forall x \in S, f(ux) = y_i \text{ if } q(u) \text{ and } x = s,$$

where  $y_i$  is the changed value of  $f$  and  $q(u)$  is some formal expression of the previous history conditions.

7. If none of the conditions holds, the value for the function does not change for the current stimulus. Capture this detail by adding the recursion rule

$$f(ux) = f(u) \text{ otherwise.}$$

8. The condition statements may make reference to as-yet undefined specification functions. These must be defined now. Mutual recursion is not a problem, provided the recursion is always to shorter sequences (in this case, to the prefix  $u$ ).

Consider the following simple example, where the numbers indicate requirements. (1)A safe uses a simple three-digit combination; these digits must be entered in the correct order to unlock the safe. (2)Following an incorrect entry, a "clear" key must be pressed before the safe will accept further input. (3)Once the three digits are entered, the safe unlocks and the door opens. (4)When the door is closed, the safe automatically locks. Let the three digits be denoted  $d_1, d_2$ , and  $d_3$ , and assume for simplicity that the digits are all distinct (this assumption can be avoided through the use of abstract stimuli, discussed later). Any digit which is not part of the combination is denoted  $d$ , the clear key is denoted  $c$ , and closing the door is denoted  $D$ . One sequence which unlocks the safe is thus  $Dcd_1dcd_1d_2d_3$ . Write a function to determine whether the door is open.

First, a descriptive name is required. Let the function be called "door." Second the domain must be specified; in this case, the domain is  $S^*$ . Third, the co-domain must be specified. For now, choose {open, closed}; the co-domain may be revisited as the function is written (for example, were it not possible to tell the state of the door, a value "unknown" could be introduced). Fourth, the special initial value rule is needed. In this case, assume the door is initially open (though we could have chosen "unknown"). This gives the rule:

$$\text{door}(\lambda) = \text{open.}$$

The fifth step is to consider conditions under which the value changes. There are two: when the door is closed, and when the third digit of the combination is entered. Step six

gives two rules (with universal quantification over  $u$  and  $x$  understood):

$$\text{door}(ux) = \text{open if } \text{combo}(u) = 2 \text{ and } x = d_3$$

$$\text{door}(ux) = \text{closed if } x = D.$$

The first rule makes use of an as-yet undefined specification function “combo” to determine the progress of combination entry. It will be stated in a moment. Step seven gives the recursive rule:

$$\text{door}(ux) = \text{door}(u) \text{ otherwise.}$$

By the same process, one can produce the following definition of the function  $\text{combo}: S^* \rightarrow \{0, 1, 2, 3, \text{error}\}$  (with some shorthand introduced for the current stimulus and universal quantification over  $u$  and  $x$  again understood):

$$\text{combo}(\lambda) = 0$$

$$\text{combo}(uD) = 0$$

$$\text{combo}(uc) = 0$$

$$\text{combo}(ud_1) = 1 \text{ if } \text{combo}(u) = 0 \text{ and } \text{door}(u) = \text{closed}$$

$$\text{combo}(ud_2) = 2 \text{ if } \text{combo}(u) = 1 \text{ and } \text{door}(u) = \text{closed}$$

$$\text{combo}(ud_3) = 3 \text{ if } \text{combo}(u) = 2 \text{ and } \text{door}(u) = \text{closed}$$

$$\text{combo}(ud_1) = \text{error if } \text{combo}(u) \neq 0 \text{ and } \text{door}(u) = \text{closed}$$

$$\text{combo}(ud_2) = \text{error if } \text{combo}(u) \neq 1 \text{ and } \text{door}(u) = \text{closed}$$

$$\text{combo}(ud_3) = \text{error if } \text{combo}(u) \neq 2 \text{ and } \text{door}(u) = \text{closed}$$

$$\text{combo}(uD) = \text{error if } \text{combo}(u) = 0 \text{ and } \text{door}(u) = \text{closed}$$

$$\text{combo}(ux) = \text{combo}(u) \text{ otherwise.}$$

Note that even though there is mutual recursion, the recursion is always on shorter sequences, and thus the basis case is eventually reached and the functions are well-defined.

Looking back over the two function definitions, it should be clear how to produce a state machine from each. Given an effective means to compute each of the component predicates, one can thus simply replace the specification function with a piece of state data whose range is the range of the function. The initial value for the state data item is given by the basis case, and the recursive rule simply states that if the state data item’s value is not changed it remains the same, and the rule can thus be dropped.

A detailed proof that the complexity of a prefix-recursive function is no greater than that of its most complex component is beyond the scope of this paper. However, consider the case in which each component predicate can be expressed by a finite state machine [4]. It is fairly easy to prove that the entire prefix-recursive function can be implemented by a Mealy machine. Further, a prefix-recursive function built only of other prefix recursive functions that have a finite-state representation can be represented by a Mealy machine.

## 8.1. Writing the Black Box as a Recursive Rule

Since  $\text{BB}(\lambda) = 0$ , one need only consider the restricted mapping  $\text{BB}|S^+$ , and this can be expressed in an equivalent form as  $\text{BB}': S^* \times S \rightarrow R$ . Thus for sequence  $u \in S^*$ , stimulus  $x \in S$ , and response  $r \in R$ ,  $\text{BB}(ux) = r$  may be read “given history of use  $u$  and the current stimulus  $x$ , the next software response is  $r$ .” Such a collection of rules may be written in tabular form, with one table for each stimulus and two columns: one column for previous history conditions (expressed using prefix-recursive functions), and one for the correct next response. (A third and fourth column for tagging and tracing should also be included. See [10].)

Written in this manner, the black box is, itself, a prefix-recursive function. The basis case ( $\text{BB}(\lambda) = 0$ ) is assumed, and need not be given explicitly. The recursive rule ( $\text{BB}(ux) = \text{BB}(u)$  otherwise) is not needed since the black box must be complete (there is no “otherwise”—it is always false). This gives a representation for the black box which is easily converted to a state machine, which has a simple, straightforward representation in tabular form, and which avoids unnecessary complication with process notations. Further, the notation for the conditions makes checking completeness and consistency a simple matter (as long as the component predicates of the prefix-recursive functions may be checked).

## 9. The Sequence Enumeration Method of Specification Writing

### 9.1. Basics of Sequence Enumeration

How does one derive such a black box specification from the requirements? A straightforward, systematic means exists in *sequence enumeration*, which is the literal enumeration in order by length of sequences in  $S^*$  and the assignment of the correct response from  $R$  to each; it is essentially an enumeration of the ordered pairs of the black box function.

Revisiting the simple safe example of the previous section, the enumeration of **Table 1** may be produced. Numbers in the “trace” column refer to requirements from the problem statement. Derived requirements are indicated with an asterisk when introduced.

**Table 1: First Safe Enumeration**

Sequence	Response	Trace
$\lambda$	0	Method
$c$	0	(5*) The safe is initially open. (6*) The safe ignores keypad input when the door is open.

**Table 1: First Safe Enumeration (Continued)**

Sequence	Response	Trace
$D$	Lock safe	(4)(5)
$d$	0	(5)(6)
$d_1$	0	(5)(6)
$d_2$	0	(5)(6)
$d_3$	0	(5)(6)
$cc$	0	(5)(6)
$cD$	Lock safe	(4)(5)
$cd$	0	(5)(6)
$cd_1$	0	(5)(6)
$cd_2$	0	(5)(6)
$cd_3$	0	(5)(6)
$Dc$	0	(7*) No external confirmation is given for combination entry other than door release.
$DD$	$\omega$	(5) (8*) It is assumed (with risk) that the safe cannot be opened by means other than combination entry.
$Dd$	0	(7)
$Dd_1$	0	(7)
$Dd_2$	0	(7)
$Dd_3$	0	(7)

Very little progress is made here. It does become clear, however, that extending the sequence  $c$  is identical to extending the sequence  $\lambda$ , in terms of future responses. This can be made explicit with the notion of sequence equivalence.

Two sequences are *equivalent* if and only if they give identical responses for all future stimuli. Formally, let  $\rho$  be an equivalence relation on  $S^*$  such that:

$$(u \equiv_{\rho} v) \Leftrightarrow \forall w \in S^+, \text{BB}(uw) = \text{BB}(vw).$$

Note that  $\text{BB}(u) = \text{BB}(v)$  is *not* required.

Given this definition, it is apparent that  $\lambda \equiv_{\rho} c$ . Since equivalent sequences always agree on responses to (non-empty) extensions, it is not necessary to extend both. Since  $\lambda$  has already been extended, one can simply note that  $c$  is equivalent to  $\lambda$  and move on. When a sequence  $u$  is noted as equivalent to a previously-enumerated sequence  $v$ ,  $u$  is *reduced* to  $v$ . A sequence which cannot be reduced to a previously-enumerated sequence is *irreducible*. Thus the sequence  $\lambda$  is always irreducible. Since equivalence relations are transitive, so is reduction, and it is reasonable to require that every reduction be to an unreduced sequence. It should also be clear that all illegal sequences are equivalent, since if a prefix is impossible, the entire sequence must be impossible. No illegal sequence need be extended.

The enumeration thus viewed forms a reduction system which is both Church-Rosser and noetherian [1], and thus every equivalence class of  $\rho$  has a unique element which is irreducible, and every sequence in  $S^*$  is equivalent to exactly one irreducible sequence.

Continuing the enumeration and noting equivalences in the new column “Equiv,” we obtain the enumeration of Table 2. Note that there are no sequences left to extend; in this case, the enumeration is *complete*, and specifies a unique response for every sequence in  $S^*$ . Thus a complete enumeration is actually a complete, consistent, and traceably correct (with respect to the requirements from which it is derived) black box specification. To see that, indeed, all sequences are mapped, consider the sequence  $cDd_1dcd_1d_2d_3$  which is not in the enumeration. The longest prefix of this sequence which does appear is  $c$ , which is equivalent to  $\lambda$ . It follows that  $cDd_1dcd_1d_2d_3$  can be reduced to  $Dd_1dcd_1d_2d_3$ . Likewise,  $Dd_1d$  can be replaced with  $Dd$ . Finally we obtain  $Dd_1d_2d_3$ , which is in the enumeration and has a response of “Release door,” which must be the response for the entire sequence.

**Table 2: Second Safe Enumeration**

Sequence	Response	Equiv.	Trace
$\lambda$	0		Method
$c$	0	$\lambda$	(5*) The safe is initially open. (6*) The safe ignores keypad input when the door is open.
$D$	Lock safe		(4)(5)
$d$	0	$\lambda$	(5)(6)
$d_1$	0	$\lambda$	(5)(6)

**Table 2: Second Safe Enumeration (Continued)**

Sequence	Response	Equiv.	Trace
$d_2$	0	$\lambda$	(5)(6)
$d_3$	0	$\lambda$	(5)(6)
$Dc$	0	$D$	(7*) No external confirmation is given for combination entry other than door release.
$DD$	$\omega$		(8*) It is assumed (with risk) that the safe cannot be opened by means other than combination entry.
$Dd$	0		(7)
$Dd_1$	0		(7)
$Dd_2$	0	$Dd$	(7)
$Dd_3$	0	$Dd$	(7)
$Ddc$	0	$D$	(2)(7)
$DdD$	$\omega$		(8)
$Ddd$	0	$Dd$	(7)
$Ddd_1$	0	$Dd$	(7)
$Ddd_2$	0	$Dd$	(7)
$Ddd_3$	0	$Dd$	(7)
$Dd_1c$	0	$D$	(9*) Pressing clear erases any combination entered (resets the safe).
$Dd_1D$	$\omega$		(8)
$Dd_1d$	0	$Dd$	(1)(7)
$Dd_1d_1$	0	$Dd$	(1)(7)
$Dd_1d_2$	0		(1)(7)

**Table 2: Second Safe Enumeration (Continued)**

Sequence	Response	Equiv.	Trace
$Dd_1d_3$	0	$Dd$	(1)(7)
$Dd_1d_2c$	0	$D$	(9)
$Dd_1d_2D$	$\omega$		(8)
$Dd_1d_2d$	0	$Dd$	(1)(7)
$Dd_1d_2d_1$	0	$Dd$	(1)(7)
$Dd_1d_2d_2$	0	$Dd$	(1)(7)
$Dd_1d_2d_3$	Release door	$\lambda$	(1)(3)

Five additional derived requirements were discovered in the process of enumeration. These derived requirements make explicit behavior which was omitted from the initial requirements.

## 9.2. Writing Black Box Tables From an Enumeration

Every sequence in an enumeration is a single-stimulus extension of some unreduced, legal sequence; every sequence in  $S^*$  is equivalent to one unreduced, legal sequence. The unreduced, legal sequences are special, and are referred to as *canonical sequences*.

Recall that the black box function could be written  $BB': S^* \times S \rightarrow R$ , splitting a sequence into "previous history conditions" and "current stimulus." The canonical sequences represent the set of previous history conditions. If these conditions could be expressed using specification functions (preferably prefix-recursive specification functions) then the enumeration could be transformed into tables very easily.

The process by which one discovers the previous history conditions defined by the canonical sequences is called *canonical sequence analysis*. The following is an outline of the process.

1. Construct a condition matrix. Rows of the matrix represent canonical sequences, and columns represent named properties of the system under consideration.
2. Extract all canonical sequences from the enumeration, and list them down the left side of the table. The canonical sequences should be listed in the order they appear in the enumeration.
3. Beginning with the first canonical sequence after  $\lambda$

(which represents initial conditions), work as follows for each canonical sequence.

- Split the sequence into prefix  $u$  and last stimulus  $x$ .
- Copy any entries in the row for  $u$  into the row for  $ux$ .
- Does processing  $x$  change the value of any listed property? If so, change the value.
- Does processing stimulus  $x$  change the value of any properties which are not listed? If so, add a column for the new property, write the value of the property in the cell for  $ux$ , and write the previous value of the property (prior to processing  $x$ ) in the cell for  $u$ .
- Continue to add property / value information until the conditions for  $ux$  are disjoint from all other conditions. If this cannot be done,  $ux$  is not canonical.
- Add any additional property / value pairs you feel are needed for clarification.

This activity is performed in Table 3 for the enumeration of Table 2.

**Table 3: Canonical Sequence Analysis**

	door	combo
$\lambda$	open	0
$D$	closed	0
$Dd$	closed	error
$Dd_1$	closed	1
$Dd_1d_2$	closed	2

The entries in the table associate a precise condition with each canonical sequence. By viewing the properties as specification functions, one can read off predicates on sequences. For example for canonical sequence  $Dd$  the condition is “door( $u$ ) = closed and combo( $u$ ) = error.”

These specification functions (one for each property) can now be generated from the enumeration and canonical sequence analysis (no further information is necessary). Though this generation process is tedious, it can be easily automated.

Let  $C_u$  denote the complete condition for canonical sequence  $u$ . Each specification function has a basis case given by its entry for  $\lambda$  in the canonical sequence analysis. For example, the entry for “door” is “open.” This gives the basis case door( $\lambda$ ) = open. Next, examine each legal sequence (except  $\lambda$ ) in the enumeration.

- For each mapping of an unreduced sequence  $ux$  in the

enumeration, compare  $C_u$  to  $C_{ux}$ .

- For each mapping of a sequence  $ux$  that is reduced to  $v$ , compare  $C_u$  to  $C_v$ .

If property  $f$  has value  $a$  for  $C_{ux}$  and value  $b$  for the condition to which it is compared, then add the following rule to the definition of the specification function:

$$\forall w \in S^*, f(wx) = b \text{ if } C_u(w).$$

Finally, add the recursive rule to every function  $f$ :

$$\forall w \in S^*, \forall x \in S, f(wx) = f(w) \text{ otherwise.}$$

Given the enumeration in Table 2 and the analysis of Table 3, definitions for specification functions “door” and “combo” are derived in Table 4 and Table 5, respectively. A comparison reveals that these specification functions are almost identical to the specification functions derived earlier, except that in the earlier version of “combo,” the combination is not reset until the door is closed.

**Table 4: Specification Function door( $ux$ )**

$$\text{door}(\lambda) = \text{open}$$

Previous History Condition	Current Stimulus	Value	Trace
door( $u$ ) = open	$D$	closed	(4)(5)
door( $u$ ) = closed and combo( $u$ ) = 2	$d_3$	open	(1)(3)
otherwise		door( $u$ )	Method

**Table 5: Specification Function combo( $ux$ )**

$$\text{combo}(\lambda) = 0$$

Previous History Condition	Current Stimulus	Value	Trace
door( $u$ ) = closed and combo( $u$ ) = 0	$d, d_2, d_3$	error	(7)
door( $u$ ) = closed and combo( $u$ ) = 0	$d_1$	1	(7)
door( $u$ ) = closed and combo( $u$ ) = error	$c$	0	(2)(7)

**Table 5: Specification Function  $\text{combo}(ux)$   
(Continued)**

$$\text{combo}(\lambda) = 0$$

Previous History Condition	Current Stimulus	Value	Trace
door( $u$ ) = closed and $\text{combo}(u) = 1$	$d, d_1, d_3$	error	(1)(7)
door( $u$ ) = closed and $\text{combo}(u) = 1$	$d_2$	2	(1)(7)
door( $u$ ) = closed and $\text{combo}(u) = 2$	$d, d_1, d_2$	error	(1)(7)
door( $u$ ) = closed and $\text{combo}(u) = 2$	$d_3$	0	(1)(3)
otherwise		$\text{combo}(u)$	Method

Finally, the black box tables must be generated (again, no information beyond the enumeration and analysis is required). The process for generating the black box is simpler than the process for deriving the specification functions, and may also be automated.

Consider every sequence (except  $\lambda$ ) in the enumeration (legal and illegal). If sequence  $ux$  is mapped to response  $r$ , then add the black box rule:

$$\forall w \in S^*, \text{BB}(wx) = r \text{ if } C_u(w).$$

The black box tables for the safe example appear as Table 6 through Table 11. In these tables,  $u$  denotes the previous stimulus history. To save space, these tables (except the first) have been compressed by merging the conditions and traces of rows with the same response.

**Table 6: Stimulus  $c$**

Previous History Condition	Response	Trace
door( $u$ ) = open	0	(5)(6)
door( $u$ ) = closed and $\text{combo}(u) = 0$	0	(7)
door( $u$ ) = closed and $\text{combo}(u) = \text{error}$	0	(2)(7)

**Table 6: Stimulus  $c$  (Continued)**

Previous History Condition	Response	Trace
door( $u$ ) = closed and $\text{combo}(u) = 1$	0	(9)
door( $u$ ) = closed and $\text{combo}(u) = 2$	0	(9)

**Table 7: Stimulus  $D$**

Previous History Condition	Response	Trace
door( $u$ ) = open	Lock safe	(4)(5)
door( $u$ ) = closed	$\omega$	(8)

**Table 8: Stimulus  $d$**

Previous History Condition	Response	Trace
any	0	(1)(5)(6)(7)

**Table 9: Stimulus  $d_1$**

Previous History Condition	Response	Trace
any	0	(1)(5)(6)(7)

**Table 10: Stimulus  $d_2$**

Previous History Condition	Response	Trace
any	0	(1)(5)(6)(7)

#### 4. Managing Sequence Enumeration Through Abstraction

While the enumeration of Table 2 does reveal information missed in the requirements, it does represent a considerable amount of effort for the result obtained. The enumeration process can be focused and controlled through the explicit use of abstraction.

A *sequence abstraction* provides an alternate view of a sequence which hides or amplifies certain details. As

**Table 11: Stimulus  $d_3$**

Previous History Condition	Response	Trace
door( $u$ ) = open or (door( $u$ ) = closed and combo( $u$ ) $\neq$ 2)	0	(1)(5)(6)(7)
door( $u$ ) = closed and combo( $u$ ) = 2	Release door	(1)(7)

developers work, they often begin to invent terminology for software conditions, such as “complete combination” or “error entry.” It is often more productive to work at with these *abstract* stimuli instead of the discrete (or *atomic* stimuli). It can also greatly shorten enumeration effort. Let  $C$  denote the sequence  $d_1d_2d_3$ , and let  $E$  denote any sequence matching the regular expression:

$$d + d_2 + d_3 + d_1(d + d_1 + d_3) + d_1d_2(d + d_1 + d_2).$$

Then the length-four sequence  $Dd_1d_2d_3$  can be expressed as the length-two sequence  $DC$ , and the 9 sequences of the enumeration which correspond to the error condition can be replaced by the single sequence  $DE$ . In fact, the introduction of these two abstract stimuli removes the need for the atomic stimuli  $d$ ,  $d_1$ ,  $d_2$ , and  $d_3$ . The entire enumeration can be reduced to that shown in [Table 12](#).

**Table 12: Third Safe Enumeration**

Sequence	Response	Equivalence	Trace
$\lambda$	0		Method
$c$	0	$\lambda$	(5)(6)
$D$	Lock safe		(4)(5)
$C$	0	$\lambda$	(5)(6)
$E$	0	$\lambda$	(5)(6)
$Dc$	0	$D$	(7)
$DD$	$\omega$		(5)(8)
$DC$	Release door	$\lambda$	(1)(3)
$DE$	0		(1)(7)
$DEc$	0	$D$	(2)(7)

**Table 12: Third Safe Enumeration (Continued)**

Sequence	Response	Equivalence	Trace
$DED$	$\omega$		(8)
$DEC$	0	$DE$	(1)(7)
$DEE$	0	$DE$	(1)(7)

#### 4.1. Properties of Sequence Abstractions

A sequence abstraction can be viewed as a mapping  $\phi: X^* \rightarrow Y^*$  from sequences of atomic stimuli  $X$  to sequences of abstract stimuli  $Y$ . The previous example reveals the three requirements for a sequence abstraction:

- **Non-increasing.** Abstract sequences must be no longer than the corresponding atomic sequences. The mapping  $\phi$  is *non-increasing* if and only if  $\forall u \in X^*, |\phi(u)| \leq |u|$ .
- **Stimulus ordering.** Abstractions must preserve the historical ordering of stimuli. The mapping  $\phi$  obeys the *stimulus ordering property* if and only if  $\forall u, v \in X^*, \phi(u)$  is a prefix of  $\phi(uv)$ . Note that this property is similar to, but weaker than, requiring that  $\phi$  be a homomorphism.
- **Disjoint.** All abstract stimuli used must be disjoint.

Not surprisingly, abstractions can be defined as prefix-recursive functions. Let  $Y = \{y_1, y_2, \dots, y_n\}$  be the set of abstract stimuli, and let  $p_1, p_2, \dots, p_n$  be a collection of mutually-disjoint predicates (where  $p_i$  is the *characteristic predicate* on  $X^*$  for abstract stimulus  $y_i$ , which is true exactly when the conditions for the abstract stimulus are satisfied). The abstraction  $\phi: X^* \rightarrow Y^*$  is thus defined as follows:

$$\phi(\lambda) = \lambda$$

$$\forall u \in X^*, \forall x \in X: \begin{cases} \phi(ux) = \phi(u)y_1 & \text{if } p_1(ux) \\ \phi(ux) = \phi(u)y_2 & \text{if } p_2(ux) \\ \dots \\ \phi(ux) = \phi(u)y_n & \text{if } p_n(ux) \\ \phi(ux) = \phi(u) & \text{otherwise.} \end{cases}$$

Clearly any function so defined obeys the stimulus ordering property. For each atomic stimulus at most one abstract stimulus is added to the abstract sequence, so any function so defined is also non-increasing.

Abstract stimuli must be disjoint; that is, two abstract stimuli cannot both apply at the same time. This is the easiest requirement to miss. Consider what would have happened had we defined  $E$  as follows:

$$(d + d_1(d + d_1 + d_3) + d_1d_2(d + d_1 + d_2) + d_2 + d_3)(d + d_1 + d_2 + d_3)^*.$$

This is a regular expression for all sequences of  $d$ ,  $d_1$ ,  $d_2$ , and  $d_3$  which do not start  $d_1d_2d_3$ , and is the “obvious” way to define  $E$ .

Given this new definition of  $E$ , consider the atomic sequence  $Dd_1dd_2dd$ . This may be mapped to abstract sequences  $DE$ ,  $DEE$ ,  $DEEE$ , or  $DEEEE$ , and thus  $\phi$  is no longer a well-defined function. Further, do we need to add  $DEE$  to the enumeration? None of the sequences to which it corresponds is illegal, but all are already covered under  $DE$ . Care must be taken to insure that all abstract stimuli used at once are disjoint.

The characteristic predicates for abstract stimuli can themselves be specified in prefix-recursive form. For example, the abstract stimulus  $E$  can be defined by characteristic predicate  $p_E$  as follows:

$$p_E(\lambda) = \text{false}$$

$$\forall u \in X^*, \forall x \in X:$$

$$\left( \begin{array}{l} p_E(ud) = \text{true if door}(u) = \text{closed} \\ p_E(ud_1) = \text{true if door}(u) = \text{closed and combo}(u) \neq 0 \\ p_E(ud_2) = \text{true if door}(u) = \text{closed and combo}(u) \neq 1 \\ p_E(ud_3) = \text{true if door}(u) = \text{closed and combo}(u) \neq 2 \\ p_E(ux) = \text{false otherwise.} \end{array} \right.$$

## 4.2. Working With Abstractions

Enumerations are constructed to explore and define system behavior. For this reason, it is seldom advisable to spend much time on precise definitions of abstract stimuli. Abstract stimulus definitions will be based on the current understanding of the system and as enumeration proceeds that understanding may change, possibly invalidating any hard work on precise, formal abstract stimulus definitions. For this reason, it is recommended that abstraction definitions be left informal until a complete enumeration is obtained. Any informal abstract stimulus definition must be sufficiently precise to check the following two properties:

- The abstract stimulus is disjoint from all other abstract stimuli.
- The definition can be unambiguously understood by all developers.

Once a complete enumeration is in hand, constructing a formal abstraction definition will typically be much easier. There are advantages to having formal definitions for abstractions: constructing a formal definition for an abstraction provides a check on the soundness of the definition, creates an artifact which can be more widely shared and reused, and provides a simple means to remove abstractions from a black box specification.

Let  $\text{BB}: Y^* \rightarrow R$  be an abstract black box and let  $\phi: X^* \rightarrow Y^*$  be the abstraction used. The atomic black box is therefore  $\phi \cdot \text{BB}: X^* \rightarrow R$ , which can be discovered by composing the abstraction with the abstract black box. In terms of prefix-recursive functions, this is simply a matter

of substituting the definition of an abstract stimulus everywhere the stimulus is mentioned. An example of this technique can be found in the case study of [12].

## 4.3. Abstract Responses

Consider the abstract enumeration of Table 12. The resulting specification is complete and consistent for the abstract stimulus set  $\{c, D, C, E\}$ . Consider, however, the atomic sequence  $Dd_1cd_1d_2$ . This sequence maps to abstract sequence  $D$ , dropping all other atomic stimuli since neither  $C$  nor  $E$  is satisfied. This should not be surprising, since abstractions are intended to hide details and, if the sequences get shorter without a sharp increase in the number of stimuli there must be some information loss. There is thus some risk associated with using abstractions. This can be countered by removing abstractions, thus exposing any incompleteness at the level of atomic stimulus sequences. Any such missed blocks are easily identified, and can then be further investigated.

## 4.4. Using Abstract Responses

An abstraction is a many-to-one mapping which may hide details. Let  $\text{BB}: S^* \rightarrow R$  be the atomic black box function, and let  $\text{BB}_Y: Y^* \rightarrow R_Y$  be an abstract view of the same black box with abstraction  $\phi: S^* \rightarrow Y^*$ .

Let  $u, v \in S^*$  be two atomic sequences such that  $\phi(u) = \phi(v)$  but  $\text{BB}(u) \neq \text{BB}(v)$ . It is no longer possible to distinguish between these two sequences to assign different responses at the abstract level, and thus a single *abstract* response  $r$  must be assigned in the abstract black box. The hidden information may be recovered from the atomic stimulus sequence (of which the abstract sequence is merely a different view), and one may thus view any such abstract value  $r$  as a mapping from atomic sequences to atomic responses  $r: S^* \rightarrow R$ .

By considering all abstract responses to be functions (abstract responses which correspond one-to-one with atomic responses may be viewed as constant functions), the relationship between the abstract and atomic black boxes can be more clearly stated:

$$\text{BB}_Y(\phi(u))(u) = \text{BB}(u).$$

(Here the abstract response is  $\text{BB}_Y(\phi(u))$ , which is actually a function. The value of this function at  $u$  is the correct atomic response.)

## 5. Conclusions

Sequence-based specification techniques provide a systematic means to discover a black box specification from behavioral requirements. The enumeration process results in an arguably complete, consistent, and traceably correct specification of external software behavior. Sequence abstraction techniques can be used to focus and control the work. The emphasis on prefix-recursive

functions throughout simplifies (and provides opportunities to automate) many of the procedures. A black box developed using these techniques is fully traceable to requirements and provides for easy transformation into a state machine (the state box).

Though the focus here was on the use of prefix-recursive functions to express the specification discovered by sequence enumeration, the enumeration and abstraction techniques are independent of the final form in which the specification is expressed. One could have used another tabular form, such as those discussed in [8], or a schema-based representation, such as the Z notation [13] to express the specification.

These techniques are currently being used on industrial applications with good results. An early version of a commercial tool which supports enumeration, canonical sequence analysis, and generation of all black box tables is available.

Sequence-based specifications are easily transformed into state machines (for details, see [9]), a fact which has led to research into developing software usage models [14] and test oracles directly from such a specification. The development and maintenance of a sequence-based software specification has many advantages which may be exploited during testing [11].

## 6. References

- [1] R.V. Book and F. Otto. *String Rewriting Systems*. Springer-Verlag, New York, New York, 1993.
- [2] M. Deck. "Development Practices." In *Cleanroom Software Engineering Practices*, S.A. Becker and J.A. Whittaker, eds. Idea Group Publishing, Harrisburg, Pennsylvania, 1997.
- [3] A.R. Hevner and H.D. Mills. "Object-oriented System Development with Box Structures." *IBM Systems Journal*, 32(2), (1993).
- [4] J.E. Hopcroft and J.E. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.
- [5] H. Mao. "The Box Structure Development Method" (Dissertation). University of Tennessee, Knoxville, Tennessee, December 1993.
- [6] H.D. Mills. "Stepwise Refinement and Verification in Box-structured Systems." *IEEE Computer*, 21(6):23-36, (June 1988).
- [7] R. Oshana. "An Industrial Application of Cleanroom Software Engineering—Benefits Through Tailoring." In *Proc. 31st Hawaii International Conference on Software Engineering*, 6:122-131, (January 1998). IEEE.
- [8] D.L. Parnas. "Tabular Representation of Relations." CRL Report No. 260, McMaster University, Hamilton, Ontario, October 1992.
- [9] S.J. Prowell. *Sequence-Based Software Specification* (Dissertation). University of Tennessee, Knoxville, Tennessee, May 1996.
- [10] S.J. Prowell and J.H. Poore. "Sequence-based Software Specification of Deterministic Systems." *Software—Practice and Experience*, 23(3):329-344, (March 1998).
- [11] S.J. Prowell. "Impact of Sequence-Based Specification on Statistical Software Testing." In *Proc. of the Second International Software Quality Week Europe*, (November 1998). Software Research Institute, Inc.
- [12] S.J. Prowell, C.J. Trammell, R.C. Linger, and J.H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley-Longman, Reading, Massachusetts. To appear: Spring 1999.
- [13] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, New York, New York, 1989.
- [14] G.H. Walton, J.H. Poore, and C.J. Trammell, "Statistical Testing of Software Based on a Usage Model." *Software—Practice and Experience*, 25(1):97-108, (January 1995).