Foundations of Sequence-Based Software Specification

Stacy J. Prowell, *Member*, *IEEE Computer Society*, and Jesse H. Poore, *Member*, *IEEE*

Abstract—Rigorous specification early in the software development process can greatly reduce the cost of later development and maintenance, as well as provide an explicit means to manage risk and identify and meet safety requirements. Sequence-based software specification is a collection of techniques for implementing rigorous, practical software specification. The primary result of this research is the sequence enumeration method of specification writing. Straightforward, systematic enumeration of all sequences to produce an arguably complete, consistent, and traceably correct specification is made practical by controlling the growth of the process.

Index Terms—Software specification, sequence-based specification, trace specification, requirements analysis, correctness, completeness, consistency.

1 Introduction

Sequence-based software specification is a collection of techniques for implementing rigorous, practical software specification. The central techniques, sequence enumeration and sequence abstraction, provide a systematic way to explore and discover intended system behavior in a complete, consistent, and traceably correct way.

Sequence-based specification was developed to facilitate the use of the Box Structure Development Method (BSDM) presented in [9]. BSDM operates by identifying and refining abstractions to develop a software system through three views: external behavior (the black box), encapsulated state (the state box), and effective procedure (the clear box). The use of sequence-based techniques to develop a black box specification gives developers additional guidance in deriving each view. For a discussion of the development of black and state box specifications without the use of sequence-based techniques, see [10].

Software behavior is specified in terms of the appropriate next externally observable response to each sequence of external inputs. (The term "trace" is sometimes used for such sequences; here, the term "trace" is reserved for use with requirements traceability.) This style of specification was examined and compared with algebraic specification in [6]. The approach taken here differs in several respects from the Trace Assertion Method (TAM) presented in [1], [11] and discussed at length in [7], and differences will be noted where appropriate. In particular, nondeterminism is allowed throughout TAM specifications, while it is discouraged in sequence-based specifications (with overspecification avoided through the use of sequence abstraction). Determinism is a special case for TAM specifications; sequence-based specifications admit small, controlled doses of nondeterminism as a means to limit complexity. TAM specifications are *descriptive* in that they describe the function of a software system, while the specifications presented here are intended to be developed in increments from imperfect starting requirements, and the restriction to determinism (at some level of abstraction) often serves to force details into the open. The specification style relies on processing sequences of events and is similar to stream processing systems (see [3], [4]).

Sequence-based specification is currently being used on large industrial projects [8], and additional refinements of both theory and practice are still being developed. A case study can be found in [14].

Section 2 presents a short introduction to black box specification appropriate for the discussion of sequence-based specification. Section 3 presents an example application of the techniques. Section 4 presents the theoretical basis for sequence enumeration, and Section 5 presents the theoretical basis for sequence abstraction. Finally, Section 6 presents an additional technique for managing the complexity of certain kinds of specifications, and Section 7 discusses future research directions.

2 Black Box Specification

The black box is the first of the three forms of specification used in BSDM. A black box specification defines a software system's behavior solely in terms of external events in as implementation-free a manner as possible. The primary application of sequence-based techniques is in the creation and maintenance of a black box specification.

The black box specification is written in terms of sequences of external events. In order to facilitate implementation, this specification is transformed into a state machine called the state box, which is the second form of specification used in BSDM. The use of sequence-based techniques to specify a black box greatly facilitates the creation of a state box specification. The third form is the implementation of the state box in a programming language.

• *The authors are with the Computer Science Department, 203 Claxton Complex, 1122 Volunteer Blvd., The University of Tennessee, Knoxville, TN 37996-3450. E-mail: sprowell@cs.utk.edu.*

Manuscript received 13 June 2001; revised 3 Sept. 2002; accepted 10 Dec. 2002.
Recommended for acceptance by T. Maibaum.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 114348.

2.1 The Black Box Function

Throughout the rest of this paper, the term *system* will refer to the collection of software components one is interested in specifying, and the term *environment* will refer to all entities external to the system with which the system (when implemented) directly communicates. The *system boundary* is the list of all system interfaces. Events (inputs, interrupts, invocations) in the environment which can affect system behavior are *stimuli*, while system behaviors which are observable in the environment are *responses*. Denote stimulus and response sets by S and R, respectively.

Let S^* denote the set of all finite-length sequences of elements of S and let λ denote the empty sequence. The black box function $\mathrm{BB} : S^* \to R$ is a complete function which maps each sequence of stimuli to the correct response. A mapping rule for this function is a *black box specification*, or just a *black box*.

The function BB is required to be a complete function; every sequence in S^* must map to some corresponding value in R. In order to deal with cases in which no externally observable response is to be generated as operational behavior for a stimulus sequence, a special value *null*, denoted 0, is introduced into R to which such sequences are mapped. For convenience in developing the theory, $\mathrm{BB}(\lambda) = 0$ is required in all cases.

Some sequences in S^* may make no sense with respect to operational behavior given the initial conditions and stimulus definitions. In terms of operational behavior, any event in a stimulus history must first be generated in the environment and then must be observed by the software. For each environment event x, let G_x be a predicate over S^* which is true exactly when environment conditions permit the environment to generate event x, let O_x be a predicate over S^* which is true exactly when the system can observe event x, whether or not the system is correctly implemented, and let C_x be a predicate over S^* which is true exactly when a correctly implemented system can observe event x. The following cases result.

- If $O_x(u) = false$ for sequence u, then the sequence ux cannot be observed by the system. Such sequences are said to be *illegal by system definition* since they violate the definition of the system. An example of such a sequence is one which contains events prior to system invocation.
- If $G_x(u) = false$ and $O_x(u) = true$ for sequence u, then the sequence ux cannot be generated by the environment even though it could be observed by the system. Such sequences are *illegal by environment definition* since they violate the definition of the environment. For example, an external system may always generate values in a particular range. Values outside this range thus contradict known properties of the environment.
- If $C_x(u) = false$ and both $O_x(u) = true$ and $G_x(u) = true$ for sequence u, then sequence ux cannot be observed by the correctly implemented software, though it can be generated and potentially observed (if there is a software fault). Such sequences are *illegal by design* since one must correctly implement code to make them impossible. For example, a graphical user interface may have items "grayed out" or hidden to make the corresponding stimuli impossible.

Sequences which are illegal by system definition are self-contradictory; there is minimal risk in ignoring these sequences during system specification because they are logically impossible. The decision that a sequence is illegal by environment definition depends on knowledge of the environment, and there is risk in ignoring these sequences. For example, the system may be ported, or environment components may turn out to be poorly understood or to fail in unexpected ways. There is obvious risk in ignoring sequences which are illegal by design. If the software is incorrectly implemented or if external software libraries or hardware systems fail, these sequences may occur. This distinction among different kinds of illegal sequences is important; often, sequences which are illegal by design are treated as if they were simply impossible, leading to unexpected behavior in practice when these supposedly impossible sequences occur. At the level of the black box one is dealing with a mathematical (as contrasted with operational) entity, and absolutely every sequence in S^* must be mapped.

The decision of whether or not to ignore illegal sequences other than those which are illegal by system definition is a risk management decision and may differ between components of a large system. It is important to document why a sequence is being considered illegal so that, if the assumptions about the system or environment change, one can track the impact on the specification and re-assess the risk.

To deal with illegal sequences, a special value *illegal*, denoted ω, is introduced into R. Illegal sequences which are to be ignored are mapped to ω by the function BB. Note that the concept of "illegal" used here is distinct from that used in [6] and [11]. The concept used in the these papers is analogous (but not equivalent) to the idea of "illegal by design."

2.2 The Black Box Partition

For a mapping $f : A \to B$, let $\ker f$ denote the equivalence relation given by $a \equiv_{\ker f} b$ if and only if $f(a) = f(b)$. This equivalence relation is the *kernel* of the mapping f. For arbitrary equivalence relation ρ on set A, let $[a]_\rho$ denote the equivalence class of $a \in A$ under ρ, and let $[]_\rho : A \to A/\rho$ denote the *natural* mapping.

A proof of the following theorem can be found in [5].

Theorem 1 (Factor Theorem). *Given a mapping $f : A \to B$ and an equivalence relation $\rho \subseteq \ker f$, there exists a unique mapping $g : A/\rho \to B$ such that $[]_\rho \circ g = f$. This is illustrated by the following commutative diagram.*

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "{[]_\rho}"'] & B \\
A/\rho \arrow[ur, "g"'] &
\end{tikzcd}$$

A black box function can be factored into a partition and a mapping using the factor theorem. Likewise, a black box

TABLE 1
Safe Controller Requirements

Tag	Requirement
1	The combination consists of three digits (0-9) which must be entered in the correct order to unlock the safe. The combination is fixed in the safe firmware.
2	Following an incorrect combination entry, a "clear" key must be pressed before the safe will accept further entry. The clear key also resets any combination entry.
3	Once the three digits of the combination are entered in the correct order, the safe unlocks and the door may be opened.
4	When the door is closed, the safe automatically locks.
5	The safe has a sensor which reports the status of the lock.
6	The safe ignores keypad entry when the door is open.
7	There is no external confirmation for combination entry other than unlocking the door.
8	It is assumed (with risk) that the safe cannot be opened by means other than combination entry while the software is running.

function can be specified by giving a partition Σ of S^* and a mapping $\Gamma : \Sigma \rightarrow R$. Let ρ_Σ denote the equivalence relation induced by the partition Σ. Then, $[]_{\rho_\Sigma} \circ \Gamma = \mathrm{BB}$.

Looking at the black box in terms of the factor theorem reveals the following: First, requiring that Σ be a partition ensures the completeness and consistency of the resulting specification. Second, for any black box function BB, one can obtain a unique black box; this will be the one for which $\Sigma = S^*/\ker \mathrm{BB}$ and will be referred to as the *minimum* black box. Third, this view allows the discussion of iterative refinement of the black box as iterative refinement of the underlying partition Σ. Fourth, there is a sense of nondeterminism within blocks of the partition which are mapped to a response set with cardinality greater than one. These observations motivate the following definition in which $\mathbb{P}_1 R$ denotes the set of all nonempty subsets of R.

Definition 1. A black box *is a tuple $\langle S, R, \Sigma, \Gamma \rangle$, where S and R are finite nonempty sets, Σ is a partition of S^*, and $\Gamma : \Sigma \rightarrow \mathbb{P}_1 R$ is a function mapping Σ to nonempty subsets of R. If $|\Gamma(B)| > 1$ for block $B \in \Sigma$, then B is a* composite *block. If the black box has any composite blocks, it is* under-determined.

By the factor theorem, Definition 1 provides enough information to determine a unique mapping from S^* to $\mathbb{P}_1 R$. This model for a black box allows several values for a given sequence, allowing refinement of a black box to the desired level of detail. One could exploit composite blocks to define a nondeterministic specification. Nondeterminism is typically used to avoid overspecification or to defer details; these concerns may be addressed by applying other principles and techniques. An under-determined black box may be thought of as constraining, but not necessarily determining, the black box function.

If the partition Σ is refined to a level of detail which permits each block to be mapped to a singleton, the resulting black box gives a single value for every sequence in S^*. This black box can be reduced to the minimum black box by combining any blocks $A, B \in \Sigma$ for which $\Gamma(A) = \Gamma(B)$.

Definition 2. A black box refinement *is a partitioning of a composite block of an underdetermined black box such that each of the resulting blocks has fewer values than the original block.*

One can exploit the idea of refinements to avoid the necessity of proving that a black box specification is complete and consistent (this difficult step is often neglected in practice). One may start with a known partition of S^* and then apply techniques which are proven to produce refinements. Thus, one is always dealing with a partition of S^*, and completeness and consistency are assured.

3 Safe Controller Example

This section presents an example application of sequence-based specification to provide motivation for subsequent sections. Sequence-based specification provides a technique (sequence enumeration) for iteratively discovering a specification from requirements and a technique (sequence abstraction) for controlling the growth of the enumeration process. These techniques and their formal underpinnings are discussed in detail in subsequent sections. Some terms used in this section (equivalence, abstraction) are defined formally in the following sections. Short, informal definitions are given here.

The following example is similar to the example presented in [13]. A large case study is additionally presented in [14].

Consider the requirements for a simple safe controller given in Table 1.

The first step in sequence-based specification is the definition of a system boundary which lists all interfaces between the system to be developed and the environment. In this case, the system boundary consists of the interfaces between the system and the external power, keypad, door sensor, and lock actuator.

Let the digits of the combination be denoted c_1, c_2, and c_3 in order. Based on the requirements, the stimuli (information crossing an interface into the system) in Table 2 can be

TABLE 2
Safe Controller Stimuli

Stimulus	Description
0-9	Digit press
C	Clear key press
D	Door closed
L	Power on with door locked
U	Power on with door unlocked

identified. These 14 stimuli could be enumerated, but enumeration of the individual distinct digits would be inefficient. It is better to change the level of abstraction by grouping all the different digits together as a single stimulus and enumerating the resulting five. Letting d denote any digit press results in a slight overabstraction (see Section 5.5). This can be corrected by introducing a predicate, defined on S^* by

$$p(u) \Leftrightarrow \exists v \in S^*, u = vc_1c_2c_3.$$

This predicate is true if the most recent entry looks like the combination. Using this predicate, the enumeration of Table 3 can be constructed. This table is produced as follows:

- Write down sequences of stimuli in order by length and in some predetermined order within a length.
- For each sequence, write down the required response (the desired externally observable behavior of the system, crossing some interface), indicating the requirement(s) that determine the response.
- If the sequence is impossible (cannot be observed by the system as implemented), then the sequence is illegal, and ω is written for its response.
- If the sequence should produce no externally observable behavior (perhaps the implementation will only update internal state), then the sequence is said to produce the null response, and 0 is written for its response.
- If a sequence u leaves the system in the same condition with respect to responses to further stimuli as a previously enumerated sequence v, then sequence u is said to be equivalent (reduced) to v, and this is noted in the equivalence column.
- Sequences which are either illegal or equivalent to a previous sequence are not extended. Sequences which are legal and not equivalent to a previous sequence are extended by each stimulus.
- Enumeration stops (it is complete) if there are no further sequences to extend.

Two derived requirements (9 and 10) are identified during this process. The identification of missing or contradictory requirements is a benefit of the sequence enumeration process.

An alternate enumeration could be constructed for a different abstraction. Let G denote entering the three digits of the combination in order. Let B denote entering the combination incorrectly, up to the first mistake. These two abstract stimuli can be defined formally as follows: Let G denote the atomic sequence $c_1c_2c_3$. (G is an abstract stimulus with characteristic predicate $G(u) \Leftrightarrow \exists v \in S^*, u = vc_1c_2c_3$.) Let B denote the collection of sequences

$$\{x|x \in S \wedge x \neq c_1\} \cup \{c_1x|x \in S \wedge x \neq c_2\}$$
$$\cup \{c_1c_2x|x \in S \wedge x \neq c_3\}.$$

These two abstract stimuli are clearly disjoint from one another and from C, D, L, and U, which will be preserved under the new abstraction (no two abstract stimuli can occur simultaneously). Using these two abstract stimuli,

TABLE 3
First Enumeration

Sequence	Resp.	Equiv.	Trace	Sequence	Resp.	Equiv.	Trace
λ	0		Method	Ldd	0		7
d	ω		9	LdC	0	L	2,7
C	ω		9	LdD	ω		8
D	ω		9	LdL	0	L	5,10
L	0		5	LdU	0	U	5,10
U	0		5	$Ldd[d,p]$	unlock	U	1,3,7
Ld	0		7	$Ldd[d,\neg p]$	0		1,2,7
LC	0	L	2,7	$LddC$	0	L	2,7
LD	ω		8	$LddD$	ω		8
LL	0	L	5,10	$LddL$	0	L	5,10
LU	0	U	5,10	$LddU$	0	U	5,10
Ud	0	U	6	$Ldd[d,\neg p]d$	0	$Ldd[d,\neg p]$	2,7
UC	0	U	6	$Ldd[d,\neg p]C$	0	L	2,7
UD	lock	L	4	$Ldd[d,\neg p]D$	ω		8
UL	0	L	5,10	$Ldd[d,\neg p]L$	0	L	5,10
UU	0	U	5,10	$Ldd[d,\neg p]U$	0	U	5,10

9. Sequences with stimuli prior to system initialization are illegal by system definition.
10. Re-initialization (power-on) makes previous history irrelevant.

TABLE 4
Second Enumeration

Sequence	Resp.	Equiv.	Trace	Sequence	Resp.	Equiv.	Trace
λ	0		Method	UG	0	U	6
G	ω		9	UB	0	U	6
B	ω		9	UC	0	U	6
C	ω		9	UD	lock	L	4
D	ω		9	UL	0	L	5,10
L	0		5	UU	0	U	5,10
U	0		5	LBG	0	LB	2,7
LG	unlock	U	1,3,7	LBB	0	LB	2,7
LB	0		1,2,7	LBC	0	L	2,7
LC	0	L	2,7	LBD	ω		8
LD	ω		8	LBL	0	L	5,10
LL	0	L	5,10	LBU	0	U	5,10
LU	0	U	5,10				

the enumeration of Table 4 can be constructed. (Note that there is one more stimulus at this level of abstraction than at the previous level.) This enumeration is shorter and is compatible with (does not contradict) the previous enumeration.

The sequences which are neither illegal nor reduced to a previous sequence are pulled out of the enumeration and assigned functions and values to distinguish among them in a process called sequence analysis. The sequence analysis of the enumeration in Table 4 is given in Table 5, with one possible set of functions and values.

Black box and specification function tables can now be derived (this process can be automated, and is discussed later in the paper). To save space, only two tables are presented: the black box tables for B (Table 6) and G (Table 7), the abstract stimuli. In these tables, w is a free variable on S^* for the sequence prefix (the sequence up to, but not including, the current stimulus).

The second enumeration does not specify the response to all atomic sequences. For example, the response to sequence Lc_1 is not specified. To discover this, the abstraction can be removed by function composition of the black box and abstraction functions. This is a relatively simple process when the functions and black box are written in the form used here. See [14] for an example of this.

The above illustrates the explicit enumeration and treatment of all possible scenarios of use, by length and order. The treatment consists of deciding and documenting what the requirements, as understood at the moment, say is the correct response in each case, as well as deciding what the requirements say about each sequence relative to all those preceding it in the enumeration relative to extension (future behavior). The process explicitly reveals which sequences need not be enumerated, which must be enumerated, and when enumeration is complete. The result is a complete, consistent, and traceably correct specification, a finite presentation of how the implementation must handle the infinite set of all possible sequences of inputs.

TABLE 5
Sequence Analysis

	Functions	
Sequence	Door	Error
λ	unknown	
L	lock	false
U	unlock	
LB	lock	true

4 Sequence Enumeration

Throughout this section it will be necessary to distinguish between two black box functions. The first is the intended function $\beta : S^* \to R$, which is the unique function the derived black box should denote when finished, given the requirements. The function β embodies a combination of the requirements and the understanding of practitioners and application domain experts. The second function is that denoted by the black box at any time during its definition or discovery, denoted BB. When finished, BB should be compatible with β. The precise meaning of the term "compatible" is given in Section 5.6; essentially, BB should agree with β, though it may be more abstract (written at a

TABLE 6
Current Stimulus B

Prefix Condition	Response	Trace
$\text{Door}(w) = \text{unknown}$	ω	9
$\text{Door}(w) \neq \text{unknown}$	0	1,2,6,7

TABLE 7
Current Stimulus G

Prefix Condition	Response	Trace
$\text{Door}(w) = \text{unknown}$	ω	9
$\text{Door}(w) = \text{lock} \wedge \text{Error}(w) = \text{false}$	unlock	1,3,7
$(\text{Door}(w) = \text{lock} \wedge \text{Error}(w) = \text{true}) \vee \text{Door}(w) = \text{unlock}$	0	2,6,7

higher level) than β, deferring some details to later in the development cycle.

4.1 Basic Sequence Enumeration

Definition 3. *Let A be a countable set and B a set. Let $f : A \to B$ be a function. An* enumeration *of f is a well-ordered listing of the pairs comprising f.*

In a sequence enumeration, one generates sequences from S^* by length. Within a particular length, sequence ordering is arbitrary, but must be a complete order. Usually, there is some alphabetic ordering imposed on the sequences. Throughout the rest of this section, the notation $u < v$ will indicate that sequence u occurs before sequence v in the ordering.

As each sequence is considered in turn, one assigns some value from R for the sequence (possibly the special values 0 or ω). It is possible, if some abstract set S is being used in place of the actual stimulus set or if some important details are as yet unknown, that it will not be possible to specify a particular value for the sequence. In this case, one should introduce a "don't know" value into R and map these sequences to this special value. Alternately, one may list each of the possible values for the sequence and the conditions (in terms of low-level or atomic stimuli, or possibly in terms of outstanding design decisions) under which each of the listed values is to be chosen. One may choose to refine the stimulus set to resolve this, or may use predicates or other techniques to refine the specification. Such a collection of possible values with associated criteria for choosing one of the listed values may be considered a single abstract value (see Section 5.4) and, thus, for the rest of this section, a single value will be assumed for every sequence in an enumeration. The mapping of a sequence u to value r in an enumeration is denoted $u \mapsto r$.

The process of enumeration provides many opportunities to discover "rules" for specifying the behavior of the system for a class of sequences. For example, one may notice that all sequences which end in a particular stimulus should be mapped to the same value, or that the same refinement criteria must be used. One can then construct a rule describing these sequences, and these rules may be used to eliminate a class of sequences from further consideration. A more powerful means for controlling growth of the enumeration process, sequence abstraction, is discussed later. This direct rule discovery and documentation is an optional part of the enumeration process, as rule discovery may be automated given a complete enumeration (see Section 4.3).

4.2 Sequence Equivalence

During sequence enumeration, one may discover an equivalence among sequences in the enumeration. Let S^+ denote $S^* \setminus \{\lambda\}$ (all sequences except λ).

Definition 4. *Two sequences $u, v \in S^*$ are* (Mealy) equivalent, *denoted $u \equiv_{\rho_{Me}} v$, if and only if $\forall w \in S^+, \beta(uw) = \beta(vw)$.*

Two sequences are equivalent if and only if their (nonempty) extensions are intended to always have the same value. This form of sequence equivalence differs from that used in [11] in that $\beta(u) = \beta(v)$ is not required. This provides the benefit that equivalences may be discovered earlier in the enumeration process. The form of sequence equivalence used here is similar to that used in [6], with the exception that here we are dealing with complete functions and do not include responses in the sequences. This avoids some subtle inconsistencies which can arise when responses are included in the sequence.

As sequences are enumerated in order and each sequence is considered, an equivalence may be discovered between the current sequence and a previously enumerated sequence. If the current sequence u is equivalent to previously enumerated sequence v, then one notes both the value for u and the equivalence to previous sequence v. In this case, u is said to be *reduced* to v. If there is no previously enumerated sequence v such that $u \equiv_{\rho_{Me}} v$, then u is said to be *irreducible*. The mapping of a sequence u to value r with equivalence to previously enumerated sequence v in an enumeration is denoted $u \mapsto r / \equiv v$.

Theorem 2 (Canonical Sequence). *For each sequence $u \in S^*$, there is a unique sequence (the "normal form") $v \in [u]_{\rho_{Me}}$ which is irreducible.*

Proof. Define relation η by $\forall u, v \in S^*, \langle v, u \rangle \in \eta$ if and only if $u \equiv_{\rho_{Me}} v$ and either $v < u$ or $v = u$. The pair $\langle S^*, \eta \rangle$ is a reduction system as defined in [2]. $\langle S^*, \eta \rangle$ has a least element λ, and reductions are always to sequences occurring earlier in the ordering. Thus, every chain of reductions must terminate (the noetherian property).

Choose sequences $u, v \in S^*$ such that $u \equiv_{\rho_{Me}} v$. Assume, without loss of generality, that either $v < u$ or $v = u$. Then, $\langle v, u \rangle \in \eta$ and $\langle u, u \rangle \in \eta$. This holds for any pair of sequences equivalent under ρ_{Me} (the Church-Rosser property).

It follows that every equivalence class has a unique irreducible element. □

The unique normal forms for the reduction system defined by ρ_{Me} are called *canonical sequences*. Note that all canonical sequences will be unreduced in an enumeration, but not all unreduced sequences are necessarily canonical. One may "miss" equivalences among sequences during enumeration. This results in more work being done; it does not make the enumeration incorrect.

4.3 Sequence Enumeration Techniques and Complete Enumeration

In the following discussion, several remarks about enumerations are stated in terms appropriate to enumeration practice. These could be stated and proven as theorems, but are obvious results.

Remark 1 (Reduction). If $u \mapsto r / \equiv v$ is in the enumeration for reduced v, then there is an unreduced sequence $w < v$ such that $u \equiv_{\rho_{Me}} w$ and, thus, only unreduced sequences are to be used in equivalences.

Remark 2 (Illegal Prefix). Any sequence which has a prefix u mapped to ω must also be mapped to ω since $\beta(u) = \omega$ implies $\forall w \in S^*, \beta(uw) = \omega$. Thus, during enumeration, if a sequence u is mapped to ω, one need not consider any extensions of u.

Remark 3 (Reduced Prefix). If, for sequence u, $u \mapsto r/ \equiv v$ is in the enumeration for unreduced $v < u$, then no sequence with proper prefix u need be included in the enumeration since $u \equiv_{\rho_{Me}} v$ implies $\forall w \in S^+, \beta(uw) = \beta(vw)$. Thus, during enumeration, if a sequence u is found to be equivalent to previously enumerated sequence v, then one need not consider any extensions of u.

Remarks 2 and 3 reveal the basic process for enumeration. To generate sequences of length $n+1$, each legal, unreduced sequence of length n is extended by every stimulus $x \in S$.

Definition 5. *An enumeration is* complete *if and only if all sequences of some length $n \geq 1$ are either reduced to a previous sequence in the enumeration or are mapped to ω.*

Theorem 3 (Complete Enumeration). *A complete enumeration specifies exactly one value for every sequence $u \in S^*$.*

Proof. From a complete enumeration the value of any sequence $u \in S^*$ can be determined as follows:

1. If u is in the enumeration, read off its value.
2. If u is not in the enumeration, find the longest proper prefix v of u present in the enumeration. Such a v must exist since λ must be extended by every stimulus in S. Additionally, such a v will either be mapped to ω, or will be equivalent to some sequence w with $w < v$. (If an equivalence is not given for v and v is not mapped to ω, then there must be sequences vx for all $x \in S$ in the enumeration and, thus, there is a longer prefix.) If v is mapped to ω, then u must also be mapped to ω. If v is equivalent to w for $w < v$, then, by Definition 4, v may be replaced with w in u and Step 1 repeated.

This algorithm must eventually terminate since λ is always mapped to 0 and cannot be reduced and, thus, a value for u will always be found. □

The immediate consequence of the complete enumeration theorem is that a complete enumeration defines a complete, consistent black box specification.

4.4 Sequence Enumeration Correctness

To ensure correctness of a sequence enumeration, every entry in the enumeration is traced to the requirements which justify the entry. If no requirements address the specific entry, then one must document a derived requirement, tag the derived requirement, and trace the entry to the new derived requirement. The new requirement should be justified based on information from domain experts and customers. The trace information will later be used to assist in black box correctness validation, as described in [14].

The process of explicit sequence enumeration helps ensure the requirements traceability of the resulting specification and provides a systematic way to discover incompleteness and inconsistency in the requirements. As a special effort to construct a correct specification, several people might concurrently perform the enumeration with an automated comparison of the results. Disagreements can then be resolved by clarifying and correcting the requirements or by documenting derived requirements. Thus, one may view sequence enumeration as part of the requirements analysis task.

4.5 Theory of Canonical Sequence Analysis

Every sequence enumeration, even an incomplete enumeration, encodes a consistent and traceably correct black box specification, which is also complete if the enumeration is complete. One may extract a black box from the enumeration by regarding the unreduced sequences as states of a Mealy machine and noted equivalences as transitions in the machine. Regular expressions may then be read from the resulting machine. The details of this process are beyond the scope of this paper, but it is worth mentioning that the resulting Mealy machine can be minimized and exploited as the minimum-state machine for simple systems. Further, this minimization can be used to identify missed equivalences; any indistinguishable states in the Mealy machine (i.e., states which can be merged without changing the machine's function) correspond to equivalent but unreduced sequences.

For most systems, developers may wish to summarize the results from the enumeration differently; the primary means to accomplish this is canonical sequence analysis.

Any incomplete enumeration may be completed by mapping all remaining unmapped sequences to ι (a special symbol denoting "incomplete") and making all such sequences equivalent to the first such sequence. This defines an equivalence class of sequences under ρ_{Me}, which is precisely the otherwise unmapped sequences. For this reason, the rest of this discussion will assume a complete enumeration.

For legal, unreduced sequence $u \in S^*$, let χ_u denote the condition to be associated with u (that is, χ_u is the characteristic predicate of $[u]_{\rho_{Me}}$). Each sequence ux is distinguished from u by a single stimulus x. If χ_u were known, then χ_{ux} could be deduced by asking how processing stimulus x "changes" the condition.

One can consider χ_λ to be the initial conditions and proceed by asking for each unreduced legal (length one) sequence x how processing stimulus x changes the initial conditions. The answer to this question is used to refine χ_λ and deduce χ_x. Next, these conditions are used to discover the conditions χ_u for unreduced legal sequences u of length 2, then length 3, etc., until conditions have been determined for all unreduced legal sequences (see Table 5).

To make the conditions more precise and to support the development of a black box and specification functions, conditions are expressed as a conjunction of atoms, where each atom is of the form "$function(w) = value$," where w is a free variable on S^* (see Table 7 for examples). One may name the functions and they will be generated when the black box is generated, as described in Section 4.6. For convenience, once a function name has been introduced in χ_u, one should give a value for that function for all conditions χ_{ux}.

Note that, since the conditions are characteristic predicates for blocks of S^*/ρ_{Me} and since all legal sequences are equivalent to one of the unreduced sequences, these conditions will define a partition of the set of all legal sequences in S^*. Thus, if one is working with the condition for sequence u, one should not proceed until χ_u is disjoint from all other conditions discovered so far. If one decides that χ_u and χ_v $(u \neq v)$ are not mutually exclusive, then one of the two sequences is not canonical, and an equivalence has been missed during enumeration.

4.6 Writing the Black Box

Once sequence analysis has been performed and the collection of conditions is available, one can immediately transform the enumeration into a black box and generate the definitions for all specification functions named during the analysis phase. It should be stressed that the complete enumeration is also a complete, consistent, and traceably correct black box, though it may not be in the most useful form for further development.

Generating the black box is straightforward. Let $ux \mapsto r$ (allowing $r = \omega$) be a mapping in the enumeration (if there is a reduction, it can be ignored here). Then, one has the black box rule $\forall w \in S^*, \mathrm{BB}(wx) = r$ if $\chi_u(w)$. (See Tables 6 and 7.)

Generating specification functions is a bit more complicated. Each specification function has a basis case given by χ_λ. If $f(w) = a$ is an atom in χ_λ, then one has $f(\lambda) = a$ as a basis case for f (see Door in Table 4). If there is no atom for f in χ_λ, then one may use $f(\lambda) = v$, introducing the value v for "not applicable" (see Error in Table 4). For each mapping in the enumeration of the form $ux \mapsto r$ (no reduction, $r \neq \omega$), compare χ_u and χ_{ux} (see LB in Table 4). For each mapping in the enumeration of the form $ux \mapsto r/ \equiv v$ $(r \neq \omega)$, compare χ_u and χ_v (see LG in Table 6). If χ_u contains an atom of the form $f(w) = a$ and the condition to which it is compared contains an atom of the form $f(w) = b$, where $a \neq b$, then one has the rule $\forall w \in S^*, f(wx) = b$ if $\chi_u(w)$. Finally, one has the recursive rule for every function $\forall w \in S^*, \forall x \in S, f(wx) = f(w)$ otherwise.

5 Sequence Abstraction

The most general form of abstraction is discussed here. Abstraction, as all aspects of sequence-based specification, can be tailored to suit a particular practice by imposing additional properties on it. Special forms of abstraction are discussed in the case study of [14].

5.1 Theory of Sequence Abstraction

Sequence-based specification supports the definition and use of abstractions to control the growth of enumerations, support scale-up, defer details without losing them, and change views of the software system.

A sequence abstraction $\phi : X^* \to Y^*$ maps sequences of atomic stimuli X to sequences of abstract stimuli Y. The terms "atomic" and "abstract" are relative to the current level of abstraction. One may, for example, define one abstraction in terms of another abstraction. This presents no difficulty since one may simply compose the mappings.

A sequence abstraction may be a change of alphabet (a monoid homomorphism) in which some symbols are dropped and others renamed. More complex sequence abstractions can be defined based on stimulus sequences, and one may consider all previous stimuli when evaluating the sequence abstraction. For example, one might be interested in whether or not an interface had timed out while waiting for input. Counting clock pulses in the stimulus sequence leads to considering very long sequences without much being learned. To avoid this, one could define an abstract "time out" stimulus and replace atomic clock pulse stimuli with this abstract stimulus. The definition of this abstract stimulus would be based on the number of clock pulses observed since a given event.

One should never think of the abstract sequence as replacing the atomic sequence; the abstract sequence is simply another view of the atomic events.

Consider the requirements for an abstraction. First, the abstract view of a sequence must be unique and must also be available for any atomic sequence. Thus, an abstraction ϕ must be a function, but need not be surjective. Abstract sequences v for which there is no corresponding atomic sequence u such that $\phi(u) = v$ are illegal by system definition.

Second, every abstract stimulus should have some meaning in terms of atomic sequences, so an abstraction should not simply add stimuli to a sequence to mark some condition. Assume marker stimuli are added by an abstraction to indicate a condition and that all atomic stimuli are kept. Then, prefixes of the abstract sequence which omit the marker stimuli are illegal (since there is no corresponding atomic sequence), but suffixes are legal. This breaks the illegal prefix principle for sequences. Every prefix of an atomic sequence should have a corresponding prefix of the abstract sequence.

Third, suppose a prefix of the abstract sequence v corresponds to atomic sequence u, and an extension of v corresponds to a proper prefix of u. Then, the order of events in the stimulus history has been changed in the abstraction, and the abstraction no longer preserves the ordering of the stimuli. Abstractions are required to preserve stimulus ordering.

These three requirements lead to the following observations. First, all prefixes of abstract sequences must "make sense;" they must correspond to some atomic sequence and vice-versa. This means it must be possible to establish a correspondence between every prefix of an abstract sequence and some prefix of the corresponding atomic sequence.

Definition 6. *The mapping $\phi : X^* \to Y^*$ satisfies the* prefix correspondence property *if and only if $v = \phi(u)$ implies that, for every prefix u' of u, $\phi(u')$ is a prefix of v and, furthermore, that every prefix v' of v is $\phi(u')$ for some prefix u' of u.*

The second observation is that abstract sequences can be no longer than the corresponding atomic sequence.

Definition 7. *The mapping $\phi : X^* \to Y^*$ is* nonincreasing *if and only if $\forall u \in X^*, |\phi(u)| \leq |u|$.*

The third observation is that the order of events in an abstract sequence $\phi(u)$ must match the order of events in the atomic sequence u. Stimulus sequences represent historical records of stimuli received, and abstractions should preserve the ordering.

Definition 8. *The mapping $\phi : X^* \to Y^*$ satisfies the* stimulus ordering property *if and only if* $\forall u, v \in X^*, \phi(u)$ *is a prefix of* $\phi(uv)$.

The prefix correspondence property is sufficient to guarantee both the stimulus ordering property and that the mapping is nonincreasing, as the following result shows.

Lemma 1. *Let $\phi : X^* \to Y^*$ be a mapping satisfying the prefix correspondence property. Then, ϕ also satisfies the stimulus ordering property and is nonincreasing.*

Proof. The stimulus ordering property follows immediately from the prefix-correspondence property. Assume ϕ violates the nonincreasing property. Then, there is some $u \in X^*$ such that $|\phi(u)| > |u|$. By the pigeon hole principle, there must be two prefixes v_1 and v_2 of $\phi(u)$ such that $v_1 \neq v_2$ and $v_1 = \phi(u')$ and $v_2 = \phi(u')$. But, by transitivity of equals, $v_1 = v_2$ and, thus, there can be no such prefixes. □

Based on Lemma 1, the following definition satisfies our earlier requirements.

Definition 9. *A* sequence abstraction *is a mapping $\phi : X^* \to Y^*$ which satisfies the prefix correspondence property.*

5.2 Defining Sequence Abstractions

For the rest of this paper, fix atomic stimulus set X and abstract stimulus set Y. An abstraction $\phi : X^* \to Y^*$ is defined by a set of computable predicates $p_y : X^* \to \{\text{true}, \text{false}\}$, one for each abstract stimulus $y \in Y$. These predicates must be defined on all atomic sequences and must be disjoint. One can then take any atomic sequence and consider the prefix consisting of the first stimulus, the first two stimuli, the first three stimuli, etc. At most one of the predicates p_y will be satisfied by each prefix. Whenever a predicate is satisfied for a given prefix p_y, one writes down the corresponding abstract stimulus y. In this way, one can derive the abstract sequence corresponding to a particular atomic sequence.

Definition 10. *Let X be a finite, nonempty set and let $Y = \{y_1, y_2, \ldots, y_n\}$ be a set for some finite $n \geq 1$. For each y_i ($1 \leq i \leq n$), let p_{y_i} be a computable predicate called the* characteristic predicate *for y_i defined everywhere on X^* and let all such predicates be pairwise disjoint (so that $p_{y_i}(u) \wedge p_{y_j}(u)$ implies $i = j$). These predicates define a function $\phi : X^* \to Y^*$ as follows:*

$$\begin{array}{rcl} \phi(\lambda) = \lambda & & \\ \forall u \in X^*, \forall x \in X, \phi(ux) & = & \phi(u)y_1 \text{ if } p_{y_1}(ux) \\ & = & \phi(u)y_2 \text{ if } p_{y_2}(ux) \\ & \vdots & \\ & = & \phi(u)y_n \text{ if } p_{y_n}(ux) \\ & = & \phi(u) \text{ otherwise.} \end{array}$$

This style of function definition is called abstraction normal form *(ANF).*

Not all functions can be written in ANF, as the following fragments illustrate. The fragment

$$\begin{array}{rcl} f(a) & = & 1 \\ f(aa) & = & 2 \end{array}$$

cannot be put in ANF because $f(a)$ is not a prefix of $f(aa)$. The fragment

$$\begin{array}{rcl} g(a) & = & \lambda \\ g(aa) & = & \text{a b} \end{array}$$

cannot be put in ANF because the additional a adds a sequence of two symbols, not a single symbol as required.

The following result will be useful to establish later results.

Lemma 2. *Let $\phi : X^* \to Y^*$ be a mapping satisfying the prefix correspondence property. Let $u \in X^*$ be a sequence and $x \in X$ a stimulus. Then, $\phi(ux) = \phi(u)v$, where $v \in Y^*$ and $|v| \leq 1$.*

Proof. By Lemma 1, $\phi(u)$ is a prefix of $\phi(ux)$, so it remains only to show that $|v| \leq 1$. Assume otherwise. Then, $|v| > 1$ and v may be written $v = yw$ for some $y \in Y$ and $w \in Y^+$. Clearly, $\phi(u)$ and $\phi(u)y$ are prefixes of $\phi(ux)$. By the prefix correspondence property, there are prefixes u_1 and u_2 of ux such that $\phi(u_1) = \phi(u)$ and $\phi(u_2) = \phi(u)y$. Obviously, one may choose $u_1 = u$. Further, since $\phi(u)$ is a prefix of $\phi(u)y$, u is a prefix of u_2. But, $\phi(u) \neq \phi(u)y$, so $u_2 \neq u$ and $\phi(u)y \neq \phi(u)yw$, so $u_2 \neq ux$. Thus, there can be no such u_2 and $|v| \not> 1$. □

Theroem 4. *Any sequence abstraction may be written in ANF.*

Proof. Let $\phi : X^* \to Y^*$ be a sequence abstraction with $Y = \{y_1, y_2, \ldots, y_n\}$. By Lemma 1, $\phi(\lambda) = \lambda$. It remains to consider $\phi(ux)$ for sequence $u \in X^*$ and stimulus $x \in X$. By Lemma 2, either $\phi(ux) = \phi(u)$ or $\phi(ux) = \phi(u)y_i$ for some $y_i \in Y$, but not both.

For each $y_i \in Y$, define p_{y_i} by:

$$\begin{array}{rcl} p_{y_i}(\lambda) = \text{false} & & \\ \forall u \in X^*, \forall x \in X, p_{y_i}(ux) & = & \text{true if } \phi(ux) = \phi(u)y_i \\ & = & \text{false if } \phi(ux) \neq \phi(u)y_i. \end{array}$$

The predicates p_{y_i} must be pairwise disjoint. If $\phi(ux) \neq \phi(u)y_i$ for any $y_i \in Y$, then $\phi(ux) = \phi(u)$ since ϕ is a complete function. Thus, ϕ can be written in ANF. □

Theorem 5. *Any function defined in ANF is a sequence abstraction.*

Proof. Let $\phi : X^* \to Y^*$ be a function defined in ANF. The first part of the prefix correspondence property can be shown by induction. Let $u, v \in X^*$ and consider $\phi(uv)$. If $v = \lambda$, then $\phi(u)$ is a prefix of $\phi(uv) = \phi(u)$. As the inductive hypothesis, assume that $\phi(u)$ is a prefix of $\phi(uv)$ for all v such that $|v| \leq n$ for some n. Then, consider vx and $\phi(uvx)$ for $x \in X$. Since ϕ is defined in ANF, it follows that $\phi(uv)$ is a prefix of $\phi(uvx)$, and, by the inductive hypothesis, $\phi(u)$ is a prefix of $\phi(uv)$. Thus, $\phi(u)$ is a prefix of $\phi(uvx)$. Since nothing has been said about u, this must be true for any u, and the first part of the prefix correspondence property is satisfied.

Let $\phi(u) = vw$ for some $v, w \in Y^*$. If $v = \lambda$, then $v = \phi(\lambda)$. As the inductive hypothesis, assume that every prefix v of $\phi(u)$ is $\phi(u')$ for some $u' \in X^*$ if $|v| \leq n$ for some n. Then, consider vy_i for some $y_i \in Y$. By the inductive hypothesis, $v = \phi(u')$ for some $u' \in X^*$ and $vy_i = \phi(u')y_i$. By the definition of ϕ, the only way this can occur is if $p_{y_i}(u'x)$ is true for some $x \in X$ and, thus, $vy_i = \phi(u'x)$. Again, since nothing has been said about w, this must be true for any prefix v of $\phi(u)$, and the second part of the prefix correspondence property is satisfied.□

Theorems 4 and 5 establish that it is necessary and sufficient for sequence abstractions to be functions which can be placed in ANF.

The key part of an ANF definition is the set of characteristic predicates p_{y_i}. The work of defining a sequence abstraction is reduced to defining one characteristic predicate p_y for each abstract stimulus $y \in Y$ and then insuring that all characteristic predicates used in a single sequence abstraction are pairwise disjoint. This is sufficient to guarantee that the resulting abstraction is well-formed and satisfies the requirements of a sequence abstraction.

There is no restriction on the characteristic predicates, other than that they must be well-defined and complete for X^*, disjoint, and computable for any sequence in X^*.

5.3 Using Sequence Abstractions

Up to this point, we have been discussing sequence-based specification as a successive partitioning of the input domain of the intended program, using either some specification language, successive refinement, sequence enumeration and canonical sequence analysis, or a combination of all three to construct a partition Σ. Likewise, abstractions may be used to refine a partition, or to create an initial partition by changing the view during enumeration.

Theorem 6 (Abstract Partitioning). *Let $\phi : X^* \to Y^*$ be a sequence abstraction, let $\{F_1, F_2, \ldots, F_n\}$ be a partition of the abstract sequences Y^* for some positive integer n, and let $E \subseteq X^*$ be a set of atomic sequences. Then, the collection of sets $E_i = \{u \in E | \phi(u) \in F_i\}$ for $1 \leq i \leq n$ is a partition of E.*

Proof. Every E_i is a subset of E. It remains to show that every sequence is in one and only one set E_i. Let $u \in E$ be an arbitrary sequence. Then, $\phi(u)$ is a unique element of Y^* and is thus in exactly one block F_i. Thus, u must be in E_i and can be in no other set of the collection, and $\{E_1, E_2, \ldots, E_n\}$ is a partition of E. □

Based on the abstract partitioning theorem, an abstract enumeration (an enumeration using some abstract alphabet) denotes a partition of the underlying atomic sequence space. Since abstractions are nonincreasing, they can be used to shorten the work required during sequence enumeration. This leads to the following simple workflow for using sequence abstraction in conjunction with sequence enumeration.

1. Enumerate sequences at the lowest reasonable level of abstraction.
2. If work stalls or is found to be unproductive, replace atomic stimuli with abstract stimuli to resolve the difficulty. Restart enumeration with the abstract stimulus set.
3. Continue to invent abstract stimuli and enumerate until a complete enumeration is obtained, or until system behavior is sufficiently understood to write a black box specification.

Sequence abstraction may be used for other purposes. One could construct a sequence abstraction to drop all stimuli except those associated with a particular interface or phase of operation. One could then enumerate the remaining stimuli to explore the chosen single aspect of system behavior.

Sequence abstractions must be defined precisely by specifying each abstract stimulus' characteristic predicate. Abstract stimulus definitions, however, will be based on current understanding of the system behavior. Thus, it is likely that the definition of abstract stimuli will evolve over time as new system behaviors are considered. For this reason, it is suggested that initial, informal definitions be constructed and used during enumeration. After system behavior is understood, the informal definitions may be replaced with formal definitions. Constructing a formal definition of an abstract stimulus will typically be much easier once a complete enumeration is in hand since one can survey the enumeration itself to determine every instance when important system properties are changed.

If informal abstract stimulus definitions are used, they should be precise enough to ensure that the predicates for all abstract stimuli used in a given enumeration are disjoint.

5.4 Abstract Values

Since many atomic sequences are mapped to a single abstract sequence, one may lose some information about the black box function. Let $\beta : X^* \to R$ be an intended black box function. Then, R denotes the *atomic values*. Let $\phi : X^* \to Y^*$ be a sequence abstraction. If there are sequences $u, v \in X^*$ such that $\phi(u) = \phi(v)$, but $\beta(u) \neq \beta(v)$, then one cannot determine a specific atomic value given only the abstract sequence, and some new *abstract value* r must be chosen to which the abstract sequence can be mapped. Let R_Y denote the set of all such abstract values with the possibility that $R \cap R_Y \neq \emptyset$ if there are abstract sequences for which a unique atomic value may be determined. A new abstract black box mapping rule $\mathrm{BB}_Y : Y^* \to R_Y$ may be constructed.

Given an element of R_Y, additional information is necessary to determine a particular element of R. This additional information can be obtained from the atomic sequence and, thus, one may view an abstract value as a mapping from atomic sequences to atomic values.

Definition 11. *An* abstract value *is a complete mapping $r : X^* \to R$, mapping atomic sequences to atomic values.*

It is now possible to restate the relationship between the abstract and atomic black boxes as $\mathrm{BB}_Y(\phi(u))(u) = \beta(u)$. This view is not incompatible with Definition 1; one may take $\Gamma([\phi(u)]) = \mathrm{range}\, \mathrm{BB}_Y(\phi(u))$, giving an under-determined black box which is then "determined" by the abstract value functions. In fact, one can compose the abstraction ϕ, the black box function BB_Y, and the response functions to generate a black box in terms of the atomic stimuli and responses. For an example of this, see the case study of [14]. This operation is greatly simplified if all functions used are written as prefix-recursive functions (see [13] for examples).

5.5 Over-Abstraction and Predicate Refinement

Suppose one has a case as described in the previous section, in which two sequences $u, v \in X^*$ are such that $\beta(u) \neq \beta(v)$ but $\phi(u) = \phi(v) = wx$ for some abstract sequence $w \in Y^*$ and $x \in Y$. Further, assume these are the only two such sequences. It seems a bit excessive to invent and use an abstract value for such a small case.

To address such cases, one may invent a predicate p on atomic sequences and then use that predicate to refine the abstract specification. For the example just given, let p be such that $p(u)$ and $\neg p(v)$. Then, we can write

$$\begin{array}{lcl} w[x,p] & \mapsto & \beta(u) \\ w[x,\neg p] & \mapsto & \beta(v). \end{array}$$

The benefit is that the predicate is only used when needed; it is a sort of "just in time" refinement technique applied during sequence enumeration (see Table 3).

One can regard the $[x,p]$ and $[x,\neg p]$ as abstract stimuli for the purpose of canonical sequence analysis, black box, and specification function generation. The predicates in these cases can then be subsumed into the predicates of the tables to join each abstract stimulus $[x,p]$ with the "more abstract" sibling x.

One may use predicates to construct a refinement with an arbitrary number of blocks. For example, assume that p and q are predicates on X^*. Then, the following is one possible three-block refinement which can be constructed: $\{[x,p], [x,\neg p \wedge q], [x, \neg p \wedge \neg q]\}$.

Completeness and consistency may be assured by checking that both the predicate and its negation are always applied to the same sequence and that the predicate is defined everywhere on X^*.

5.6 Compatible Specifications

Definition 12. *Let* $\mathrm{BB}_Y = \langle Y, R, \Sigma_Y, \Gamma_Y\rangle$ *and* $\mathrm{BB}_Z = \langle Z, R, \Sigma_Z, \Gamma_Z\rangle$ *be black boxes with the same value set* R. *Then,* BB_Z *is an* abstraction *of* BB_Y, *and* BB_Y *is a* refinement *of* BB_Z *if and only if there exists a sequence abstraction* $\phi : Y^* \to Z^*$ *such that*

$$\forall u \in Y^*, \Gamma_Y([u]_{\Sigma_Y}) \subseteq \Gamma_Z([\phi(u)]_{\Sigma_Z}).$$

The responses prescribed by BB_Z must not preclude those given by the refinement BB_Y. This leads to the notion of compatible black boxes: Two black boxes are compatible if they can be implemented (refined) in the same way.

Defintion 13. *Two black boxes* BB_Y *and* BB_Z *are* compatible *if and only if there exists a black box* BB_W *such that* BB_W *is a refinement of both* BB_Y *and* BB_Z.

6 Interrupts

Let O denote the set of all discrete simultaneous software outputs. That is, the elements of O are tuples, with one entry for each system interface from which output may be observed. Each element of the tuple represents a required value on the particular interface; if no value is required on the interface, a "null" may be used, denoted ε. For example, a system might have n serial lines attached to it, and each serial line may take on a value in the interval $[0, 255]$. Then, one might have $O = ([0,255] \cup \{\varepsilon\})^n$. A single response $r \in R$ may consist of a sequence of such outputs; that is, $R \subseteq O^*$. Note that $\lambda \in R$ and, thus, one can also think of the null response as λ. For example, the response "print the document" may, in fact, consist of a very long sequence of outputs.

This section deals with *interrupts*, an extension to the sequence enumeration process which addresses the use of output sequences. Interrupts need only be used when the output sequence cannot be viewed atomically, that is, when the sequence may be interrupted or when it may become delayed. Though interrupts introduce nondeterminism into the enumeration, they may be viewed as abstract stimuli whose characteristic predicates are constrained, but not fixed. Thus, none of the techniques of sequence enumeration, analysis, and generation need be changed to accommodate interrupts.

Consider the safe controller from Section 3, with the following modification: If door open is detected when the safe should be locked (denoted i for intrusion), a warning beep will sound. After 20 seconds, the warning beep will change to an alarm and the police will be summoned unless the correct combination is entered first.

This problem can be addressed by sequence enumeration; it requires the invention of a "timeout" abstract stimulus defined in terms of clock pulse stimuli, countdown timers, or some such low-level stimuli. Reference to these low-level stimuli can be avoided by using an informal (but precise) definition for the timeout stimulus. Using T to denote the passage of 20 seconds, one can construct the following enumeration fragment.

$$\begin{array}{rcl} L\ i & \mapsto & \text{warn} \\ L\ i\ T & \mapsto & \text{alarm} \\ L\ i\ G & \mapsto & \text{stop warn}/ \equiv U \\ L\ i\ T\ G & \mapsto & 0/ \equiv L\ i\ T. \end{array}$$

Alternately, one may think of the i stimulus as leading to a sequence of outputs: 20 seconds of warning beeps followed by a steady alarm. This sequence may be interrupted by entering the correct combination during the warning beeps. One might then write $L\ i \mapsto \text{warn! alarm}$, where the exclamation mark indicates that the response warn may be *interrupted*.

Extensions of Li assume that the response goes to completion, that is, that the alarm is eventually issued. In addition to the usual extensions, one adds all extensions of $Li\,[\text{warn}]$, as well. These indicate that the response warn was generated, but that the sequence was interrupted so the alarm was never generated. Thus, one would write $Li\,[\text{warn}]G \mapsto \text{stop warn}$. To indicate that some other stimulus Q is not intended to interrupt the output sequence, one writes $Li\,[\text{warn}]Q \mapsto 0/ \equiv L\ i$.

One may view $[\text{warn}]G$ and $[\text{warn}]Q$ as abstract stimuli (G interrupting a warn and Q interrupting a warn) and may include them in the general stimulus alphabet for the purpose of sequence analysis and black box generation.

Thus, one would have rules in the black box for these stimuli. These allow one to specify the timeout behavior without saying anything about how the interval is actually measured. The enumeration *could* be made deterministic (and the software will be), but this would have required additional details about how the timeout interval is measured.

This approach may be generalized as follows: Let $a, b, c \in O^*$ be sequences of outputs. Then, to extend $u \mapsto a!\ b!\ c$, add sequences $u\ [a]x$ and $u\ [a\ b]x$ for all $x \in S$ to the enumeration. If u is not reduced, then the "usual" extension sequences $\{ux \mid x \in S\}$ must also be added. Finally, it is worth noting that $u\ [a\ b]x \mapsto 0/ \equiv u$ is acceptable; when writing the equivalence, one need not try to preserve the information that responses a and b have been issued since $[a\ b]x$ constitutes a single abstract stimulus.

7 Summary and Future Directions

Following Mills et al. [10] prescription of software development as the progressive consideration of requirements, then the sequences of inputs, then the invention and maintenance of state, and, finally, efficient procedures, we have looked for effective ways of deriving the sequence-based specification and then the state-based specification. The work flow from typical statements of requirements to precise specifications suitable for software development has been captured in the sequence enumeration method illustrated in Section 3 and more fully in [14]. Our experience with the method, working with hundreds of software developers in a variety of applications, indicates that it is quite efficient because it reveals issues and problems so early in the development process that it essentially eliminates rework once code development has begun.

Control of combinatorial growth of the enumeration process is essential to effective use of the method. This control is achieved by working at a suitable level of abstraction with respect to the input space, output space, and derivation of the mapping rule of the black box function. It is achieved by recognition at the earliest possible moment that a sequence is equivalent to an earlier sequence in the enumeration and by otherwise systematically eliminating sequences from further extension. Finally, control is achieved by isolation of concerns which permits partitioning, reducing the number of sequences to be considered.

Because of the generally inadequate nature of requirements, as enumeration forces discovery and decisions it necessitates starting the enumeration over or shifting levels of abstraction. In order to understand and maintain control, the process has been embedded in the theory of functions on regular expressions and finite state machines.

The theory and practice is based on discovery and maintenance of partitions of S^*. All work rules are based on changing domains, ranges, mapping rules, characteristic predicates, and partitions. This is the theory of finite state machines and sequence recognizers. Here, it is applied to deriving precise software specifications from ordinary requirements statements. Shifts in abstractions, nondeterminism, and special cases such as interrupts are handled by functions rather than less-structured relations.

If one follows the rules, the underlying mathematics need not be visible in the work flow. Moreover, the rules, applied uniformly, lend themselves to tool support.

The need for thorough testing of software leads inevitably to automated testing. Automated testing is best done when there was design for testability and, thus, a precise specification. Both theory and practice indicate that sequence-based software specification can contribute significantly to automated testing of software. When performing automated testing of a system, it may be impractical to distinguish between particular responses. For responses $q, r \in R$, let $q \equiv_\rho r$ if the automated test system cannot distinguish between response q and response r. Then, let $R_T = R/\rho$ denote the set of externally observable, distinguishable responses.

This provides one formal view linking the specification with automated testing. Denote the observed software function by $\mathrm{BB}_P : S^* \to R_T$. Then, the software is correct (as far as the automated test environment can distinguish) precisely when the following diagram commutes.

$$\begin{tikzcd}
S^* \arrow[r, "\mathrm{BB}"] \arrow[dr, "\mathrm{BB}_P"'] & R \arrow[d, "{[\,]_\rho}"] \\
& R_T
\end{tikzcd}$$

For sequence u, one can observe $\mathrm{BB}_P(u)$ and the specification will reveal $\mathrm{BB}(u)$. For an automated testing environment, the *oracle problem* can be stated as $\mathrm{BB}(u) \in \mathrm{BB}_P(u)$. Research is currently under way on applying these ideas to automatically develop oracles for software testing.

It is also possible to exploit the state machine developed from the black box as a first pass at the state machine for a Markov chain usage model [15]. This connects the theory and practice of sequence-based specification to the theory and practice of model-based testing using Markov chain usage models [12].

References

[1] W. Bartoussek and D. L. Parnas, "Using Assertions about Traces to Write Abstract Specifications for Software Modules," *Proc. Second Conf. European Cooperation in Informatics and Information Systems Methodology,* pp. 211-236, 1978.

[2] R.V. Book and F. Otto, *String Rewriting Systems.* Springer-Verlag, 1993.

[3] M. Broy, *Functional Specification of Time Sensitive Communicating Systems.* Springer-Verlag, 1990.

[4] M. Broy and C. Dendorfer, "Modelling Operating System Structures by Timed Stream Processing Functions," *J. Functional Programming,* vol. 2, no. 1, pp. 1-21, Jan. 1992.

[5] P.M. Cohn, *Universal Algebra,* second ed. Springer-Verlag, 1979.

[6] D.M. Hoffman and R. Snodgrass, "Trace Specifications: Methodology and Models," *IEEE Trans. Software Eng.,* vol. 14, no. 9, pp. 1243-1252, Sept. 1998.

[7] R. Janicki and E. Sekerinski, "Foundations of the Trace Assertion Method of Module Interface Specification," *IEEE Trans. Software Eng.,* vol. 27, no. 7, pp. 577-598, July 2001.

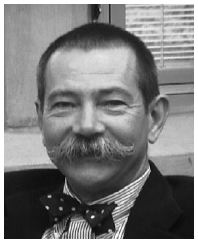
[8] D.P. Kelly and J.H. Poore, "From Good to Great: Lifecycle Improvements Can Make the Difference," *Cutter IT J.,* vol. 13, no. 2, pp. 7-14, Feb. 2000.

[9] H.D. Mills, "Stepwise Refinement and Verification in Box-Structured Systems," *Computer,* vol. 21, no. 6, pp. 23-26, June 1988.

[10] H.D. Mills, R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design,* Orlando, Fla. : Academic Press, 1986.

[11] D.L. Parnas and Y. Wang, "The Trace Assertion Method of Module Interface Specification," Technical Report 89-261, Dept. of Computing and Information Science, Queen's Univ. at Kingston, Ontario, Canada, Oct. 1989.
[12] J.H. Poore and C.J. Trammell, "Application of Statistical Science to Testing and Evaluating Software Systems," *Statistics, Testing, and Defense Acquisition: Background Papers,* M.L. Cohen, D.L. Steffey, and J.E. Rolph, eds., chapter 3, pp. 124-170, 1999.
[13] S.J. Prowell, "Developing Black Box Specifications through Sequence Enumeration," *Proc. Harlan Mills Colloquium,* May 1999.
[14] S.J. Prowell, C.J. Trammell, R.C. Linger, and J.H. Poore, *Cleanroom Software Engineering—Technology and Process.* Addison-Wesley-Longman, 1998.
[15] J.A. Whittaker and J.H. Poore, "Markov Analysis of Software Specifications," *ACM Trans. Software Eng. and Methodology,* vol. 2, no. 1, pp. 93-106, Jan. 1993.

Stacy J. Prowell received the PhD degree in computer science from the University of Tennessee in 1996. He is a research associate professor of computer science at the University of Tennessee, Knoxville. His current research interests include rigorous software specification, model-based software testing, and the development of CASE tools to support the application of rigorous software engineering methods. He is a member of the IEEE Computer Society, the ACM, Sigma Xi, and the AAAS.



Jesse H. Poore received the PhD degree in Information and Computer Science from Georgia Tech, Atlanta. He is a professor of computer science at the University of Tennessee and holds the Ericsson-Harlan D. Mills Chair in Software Engineering. He is also director of the UT-Oak Ridge National Laboratory Science Alliance. He conducts research in the economical production of high quality software. His research has been funded and driven by the software engineering needs of Ericsson, IBM, Raytheon, Nortel, and other corporations, as well as the US Army, Air Force, and Navy. He has served on many government advisory committees and panels of the National Academy of Science, most recently the NAS/NRC Panel on Statistical Methods for Testing and Evaluating Defense Systems. Dr. Poore received the IEEE Computer Society 2002 Harlan Mills Award. He is a member of the IEEE, IEEE Computer Society, ACM, and a fellow of the AAAS.