An Algorithm for Forward Reduction in Sequence-Based Software Specification

Lan Lin, Yufeng Xue
Ball State University
Department of Computer Science
Muncie, IN 47396, USA
{llin4, yxue2}@bsu.edu

Abstract

Sequence-based software specification is a rigorous method for deriving a formal system model based on informal requirements, through a systematic process called sequence enumeration. Under this process, stimulus (input) sequences are considered in a breadth-first manner, with the expected system response to each sequence given. Not every sequence needs to be further extended by the enumeration rules. The completed specification encodes a Mealy machine and forms a basis for other activities including code development and testing. This paper presents a forward reduction algorithm for sequence-based specification. The need for such an algorithm has been identified by field applications. We used the state machine as an intermediate tool to comprehend and analyze all change impacts resulted from a forward reduction, and used an axiom system for its development. We illustrate the algorithm with a symbolic example, and report a larger case study from published literature in which the algorithm is applied. The algorithm will prove useful and effective in deriving a system-level specification as well as in merging and combining partial work products towards a formal system model in field applications.

1 Introduction

Modern software development processes for safety- and mission-critical systems rely on rigorous methods for code development and testing to support dependability claims and assurance cases that provide the justified and needed confidence [14]. *Sequence-based software specification* [24, 26, 23] is such a rigorous method developed by the University of Tennessee Software Quality Research Laboratory (UTK SQRL) in the 90's to derive a formal specification and system model from the original descriptions of functional requirements. The specification, developed at an early stage in the life cycle, is important for later phases including code development, testing, and functional formal verification. Since its inception the method has been successfully combined with a rigorous testing method (Markov chain usage-based statistical testing) and applied to a variety of industry and government projects ranging from medical devices to automotive components to scientific instrumentation, to name a few [6, 5, 13, 25, 7, 26].

Sequence-based specification systematically examines the behavior of software in all possible scenarios of use (use cases) through *sequence enumeration* [18, 24, 26, 23]. In this process a human specifier, a developer, or a domain expert considers sequences of stimuli in a breadth-first manner, where sequences of a given length are examined in lexicographic order. For each such sequence they give the last output produced by software in response to the sequence of inputs, based on the requirements. There are two situations in which a sequence need not be further extended: either it is illegal (it cannot be applied in practice) or it reaches a system state equivalent to one reached by a shorter or lexicographically earlier sequence. Sequence enumeration stops when no sequence needs to be extended and results in a table of sequences that defines a Mealy machine [18, 24]. A significant benefit of the method is that it results in a formal specification but without requiring the developer to use a formal notation. This specification can form the basis for other activities including automated model-based testing. It can also be automatically analyzed to determine whether it has certain expected, or desirable properties; this can result in requirements errors being found.

However, the specification is usually not produced in one single pass but instead in iterations. Enumerated sequences are periodically reviewed, compared with each other, added, deleted, or modified against the requirements, as one gains more and better knowledge and understanding of the system under specification. It usually results in a lot of rework when a sequence is identified to have taken the system to the same state reached by a later sequence in length-lexicographical order that has been extended. In such cases all the specification work done in specifying the

DOI reference number: 10.18293/SEKE2016-192

state reached by the later sequence, as well as all its successor states, will have to be erased while the same amount of work needs to be performed on the earlier sequence due to the enumeration rules. An algorithm that enables such a "forward reduction" was needed both in theory and in practical applications to maximize automation support, preserve specification effort, and eliminate the need for a lot of rework. Field applications had also identified the need for such an algorithm as we merged or combined partial work products that focus on different system boundaries to build system models for larger and more complex applications. This paper presents a forward reduction algorithm that we have developed for this purpose. With the new theory and accompanying tool support, more requirements and specification changes can be accommodated, and the change impacts automatically computed, analyzed, and applied across the specification. The result is an enhanced specification process.

This paper is structured as follows. In Section 2 we introduce the sequence-based specification. Section 3 describes our forward reduction algorithm with a running example. In Section 4 we describe how the algorithm was applied in a published case study and report our results. Finally, Section 5 summarizes related work and Section 6 concludes the paper.

2 Sequence-based software specification

Sequence-based software specification [24, 26, 23] is a rigorous method that systematically converts ordinary functional requirements of software to a precise specification. The specification automatically translates to a formal system model for both development and testing.

To apply the method, one first identifies a *system boundary* that defines what is inside and outside the software-intensive system. This usually consists of a list of *interfaces* between the system and the *environment* of the software. From the interfaces one further collects stimuli (inputs) and responses (outputs). *Stimuli* refer to events (inputs, interrupts, invocations) in the environment that can affect system behavior. *Responses* refer to system behaviors observable in the environment. Then one explicitly *enumerates* finite stimulus sequences (representing scenarios of use), first in increasing order of length, and within the same length lexicographically. The sequence enumeration process is guided by a few rules.

As an example let us consider a simple industrial "cooker" [1]. Requirements for the cooker controller are collected in Table 1, with each sentence numbered (tagged) for easy reference. The last three requirements whose tags begin with "D" correspond to derived requirements not originally given but identified in the specification process. We further identify all stimuli and responses in Table 2. Notice the last two responses introduced by the theory: the *null* response (denoted 0) representing no observable behavior across the system boundary (the software may only have an internal state update), and the *illegal* response (denoted ω) representing a sequence of inputs not practically realizable (an instance of this is the power-on event being placed after other inputs in the sequence).

Table 1. Cooker controller requirements

Tag	Requirement
0	Consider an industrial "cooker" that is subject to various factors (external heat, internal chemical reactions, etc.) that change the pressure in the cooker. A control unit attempts to keep the pressure near a specific set point by opening and closing a pressure release valve and turning a compressor on and off, according to the following rules.
1	When started, the controller does not know the status of the valve or compressor, and there is no hardware capability to poll their status.
2	Only one output signal can be sent per input signal.
3	If both the valve and the compressor appear to need changes at the same time, since only one signal can be sent, function the valve first.
4	If the input signal says the pressure is Low, then the valve should be closed and the compressor on.
5	If the input signal says the pressure is Good, then the valve should be closed and the compressor off.
6	If the input signal says the pressure is High, then the valve should be open and the compressor off.
7	When turned off, the controller does not have time to generate any output signal.
D1	Sequences with stimuli prior to system initialization are illegal by system definition.
D2	When started, the controller does not produce any externally observable response across the system boundary.
D3	Once started, the system cannot be started again without being turned off first.

Table 3 shows a sequence enumeration of the cooker controller up to Length 3. We start with the empty sequence λ, and proceed from Length n to Length $n+1$ sequences (n is a non-negative integer). Within the same length we enumerate sequences in lexicographical order. For each enumerated sequence the human specifier makes two decisions based on the requirements:

1. They identify a unique response for the most current stimulus given the complete stimulus history. For instance, the sequence SG corresponds to: software started, followed by a pressure good reading. By Requirements 1, 2, 3, and 5, the valve should be func-

Table 2. Cooker controller stimuli and responses

Stimulus / Response	Short Name	Description	Requirement Trace
Stimulus	S	Started	1
Stimulus	L	Pressure low	4
Stimulus	G	Pressure good	5
Stimulus	H	Pressure high	6
Stimulus	O	Turned off	7
Response	ov	Open valve	0
Response	cv	Close valve	0
Response	co	Compressor on	0
Response	cf	Compressor off	0
Response	0	Null	Method
Response	ω	Illegal	Method

tioned first and closed, hence the response "cv". A sequence is *illegal* if its mapped response is ω; otherwise, it is *legal*.

2. They consider if the sequence takes the system to the same situation that has been encountered and explored by a previously enumerated sequence. If so they note the previous sequence in the "Equivalence" column, and treat the later sequence as *reduced* to the earlier sequence. For instance, the sequence SO corresponds to: software started and then turned off. This takes the system to the same situation as the empty sequence as in both cases software is not started yet, hence SO is reduced to λ. From this point on SO and λ will behave the same for any future non-empty sequence of inputs. A sequence is *unreduced* if no prior sequence is declared for its "Equivalence" column; otherwise, it is *reduced*. In theory we also treat an unreduced sequence as reduced to itself.

The enumeration process is constrained by the following rules:

1. If a sequence (say u) is illegal, there is no need to extend u by any stimulus, as all of the extensions must be illegal (i.e., physically unrealizable). For instance, no extensions of the sequence G are enumerated because G is an illegal sequence.

2. If a sequence (say u) is reduced to a prior sequence (say v), there is no need to extend u, as the behaviors of the extensions are defined by the same extensions of v. For instance, no extensions of the sequence SO are enumerated because SO is reduced to λ.

3. When reducing a sequence (say u) to a sequence (say v), we require v be both prior (in length-lexicographical order) and unreduced (otherwise we could follow the reduction chain of v and get to the sequence that is unreduced). For instance, SGO is reduced to λ and not SO.

Table 3. Cooker controller sequence enumeration up to Length 3

Sequence	Response	Equivalence	Trace
λ	0		Method
G	ω		D1
H	ω		D1
L	ω		D1
O	ω		D1
S	0		D2
SG	cv		1, 2, 3, 5
SH	ov		1, 2, 3, 6
SL	cv	SG	1, 2, 3, 4
SO	0	λ	7
SS	ω		D3
SGG	cf		5
SGH	ov	SH	2, 3, 6
SGL	co		4
SGO	0	λ	7
SGS	ω		D3
SHG	cv	SG	2, 3, 5
SHH	cf		6
SHL	cv	SG	2, 3, 4
SHO	0	λ	7
SHS	ω		D3

Therefore, only legal and unreduced sequences of Length n get extended by every stimulus for consideration at Length $n+1$. The process continues until all sequences of a certain length are either illegal or reduced to prior sequences. The enumeration becomes *complete*. This terminating length is discovered in enumeration, and varies from application to application. The cooker controller enumeration terminates at Length 5.

Application of the method is facilitated with two tools developed by UTK SQRL: Proto_Seq [2] and REAL [3]. To produce a specification in either tool, one only needs to give stimuli and responses short names to facilitate enumeration; no other notation or syntax is required. The tools enforce enumeration rules by the recommended workflow and maintain internal files (XML format) current with every action.

Table 4. An example complete and finite enumeration $\mathcal{E}_{ex}$

Sequence	Response	Equivalence
λ	0	
a	r_1	
b	0	
c	0	a
aa	0	λ
ab	r_2	
ac	r_3	λ
ba	0	ab
bb	r_3	b
bc	0	b
aba	0	ab
abb	r_1	ab
abc	r_2	ab

3 An algorithm for forward reduction

Suppose we have a complete and finite enumeration $\mathcal{E}$: *complete* because all the sequences that are both legal and unreduced have already been extended by every single stimulus, and *finite* because the total number of enumerated sequences in $\mathcal{E}$ is finite. In Table 4 we show an example of such a complete and finite enumeration, called $\mathcal{E}_{ex}$, that is purely symbolic. We ignore for now the semantics associated with application domains and functional requirements just to focus on how the algorithm is derived and how it works on a concrete enumeration example. Notice that $\mathcal{E}_{ex}$ has a stimulus set $S = \{a, b, c\}$ and a response set $R = \{r_1, r_2, r_3, 0, \omega\}$ (by theory the response set always contains the two special responses 0 and ω).

Suppose we have two enumerated sequences in $\mathcal{E}$ that are both legal and unreduced: u and v, and v is prior to u in length-lexicographical order. Because it is a complete enumeration u and v must have been extended by every single stimulus. The extensions, if they are also legal and unreduced, must have been further extended. Depending on whether the prior sequence v is the empty sequence λ or not, we handle two different cases.

In Case 1 v is not the empty sequence. An example of this is u being Sequence b and v being Sequence a in $\mathcal{E}_{ex}$. Now we want to "forward reduce" Sequence v to Sequence u based on two understandings: (1) u and v should take the system to the same situation (state), and (2) u has been extended to find this particular state and its successor states and we hope to utilize this information we have obtained. Of course our enumeration rules do not allow us to reduce v to a sequence that comes afterwards in length-lexicographical order (namely u). What we hope to achieve is to simulate the effect of a forward reduction as if we had extended v correctly (instead of u) to identify this state and all its successor states. The question is: how could we design an algorithm that automatically computes and applies all required changes to the existing specification (enumeration) to make it happen?

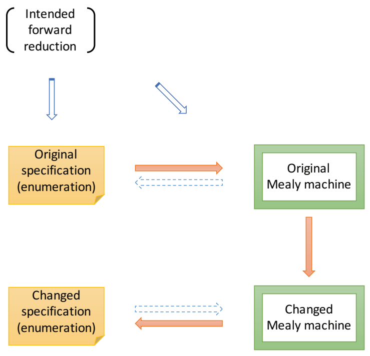

Intended forward reduction
Original specification (enumeration)
Original Mealy machine
Changed specification (enumeration)
Changed Mealy machine

Figure 1. Our approach underlying the forward reduction algorithm

As discussed in [17] a single atomic specification change (e.g., changing the response or equivalence of an enumerated sequence) could have a rippling effect on most enumeration entries and hence have a comprehensive change impact on the existing specification. The crux we found is to understand the changes utilizing state machines as an intermediate and visualizing tool, as the rigorous specifications in enumeration form have a formally defined correspondence with state machines [18], and the change impacts are more intuitively analyzed in state machines than in specifications. Figure 1 illustrates our approach.

We first give our algorithm for Case 1, and then explain it with an example.

Algorithm *CompMapForForwardRed1*($\mathcal{E}$, u, v)

Input:

A complete and finite enumeration $\mathcal{E}$, two enumerated legal and unreduced sequences u and v such that v is prior to u and $v \neq \lambda$

Output:

A hash map κ mapping each unreduced sequence in $\mathcal{E}$ to an unreduced sequence in $\mathcal{E}'$ that represents the same state (if the state is preserved after the change), or nil (otherwise)

```
1. Initialize an empty hash map $\kappa$.
2. $\kappa(\lambda) \leftarrow \lambda$
3. $i \leftarrow 0$
4. repeat
5.     for every enumerated sequence in $\mathcal{E}$ of the form:
           prefix sequence $p$ followed by stimulus $s$ is reduced to
           sequence $w$
```

```
6.          do if $ps \neq v$ and $w \neq u$ and $\kappa(w) = nil$ and
                 $\kappa(p) \neq nil$ and BlackBox($\mathcal{E}, \kappa(p)) \neq \omega$
7.              then Let $\kappa(p)$ concatenated with $s$ be
                     a candidate for $\kappa(w)$.
8.    for every unreduced sequence $w$ that has designated
          candidates in steps 5-7
9.        do $\kappa(w) \leftarrow$ its first candidate in (length-)lexicographic
             order
10.   $i \leftarrow i + 1$
11.   if $i = |v|$
12.       then $\kappa(u) = v$
13. until The last iteration has no new sequence defined for $\kappa$.
14. return $\kappa$
```

Algorithm *ForwardReduction1*($\mathcal{E}$, u, v)

Input:

A complete and finite enumeration $\mathcal{E}$, two enumerated legal and unreduced sequences u and v such that v is prior to u and $v \neq \lambda$

Output:

A complete and finite enumeration $\mathcal{E}'$

```
1. Initialize an empty hash map $\kappa$.
2. $\kappa \leftarrow$ CompMapForForwardRed1($\mathcal{E}, u, v$)
3. Initialize $\mathcal{E}'$ to contain $\lambda$ only, with $\lambda$ mapped to 0 and
   unreduced.
4. Add sequence $v$ in $\mathcal{E}'$. Map $v$ to its response in $\mathcal{E}$ and reduce
   it to $v$.
5. for every enumerated sequence in $\mathcal{E}$ of the form:
      prefix sequence $p$ followed by stimulus $s$ mapped to
      response $r$ and reduced to sequence $w$
6.    do if $ps \neq v$ and $\kappa(p) \neq nil$ and BlackBox($\mathcal{E}, \kappa(p)) \neq \omega$
7.       then Add the following sequence in $\mathcal{E}'$:
              prefix sequence $\kappa(p)$ followed by stimulus $s$ mapped
              to response $r$ and reduced to sequence $\kappa(w)$
8. for every enumerated illegal and unreduced sequence $w$ in
      $\mathcal{E}$
9.    do if $\kappa(w) \neq nil$ and BlackBox($\mathcal{E}, \kappa(w)) \neq \omega$
10.      then for every stimulus $s$
11.          do Add the following sequence in $\mathcal{E}'$:
             prefix sequence $\kappa(w)$ followed by stimulus $s$
             mapped to $\omega$ and reduced to sequence $\kappa(w)$
12. return $\mathcal{E}'$
```

A crucial step in the solution is figuring out how the state space has changed from the old automaton to the new automaton with the intended forward reduction, and what the changes have implied for the corresponding specification. As established in [18, 17] a complete and finite enumeration encodes a finite state automaton with Mealy outputs (a Mealy machine) that can be computed algorithmically. Every state of the Mealy machine corresponds to a block (equivalence class) of stimulus sequences; they all take the system to this common state starting from the initial state. Every state is represented by a unique unreduced sequence, which is the first sequence in length-lexicographical order in this block of sequences and the one you first encountered in sequence enumeration. For instance with $\mathcal{E}_{ex}$ the automaton has four states represented by unreduced sequences λ, a, b, and ab, respectively.

The main algorithm, *ForwardReduction1*, calls two algorithms: *CompMapForForwardRed1* and *BlackBox* (*CompMapForForwardRed1* also calls *BlackBox*), to compute the new enumeration after the change. We have "1" appended at the end of the names indicating Case 1 ("2" will be used for versions solving Case 2). Notice that *BlackBox* is an algorithm that computes the mapped response for any possible stimulus sequence, no matter whether it is explicitly enumerated in the table [18, 17], once you have a completed sequence enumeration.

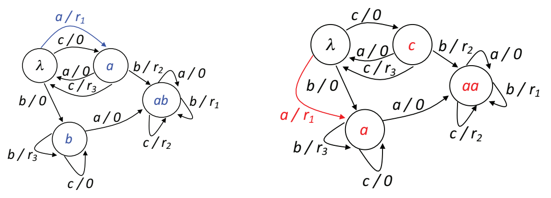
CompMapForForwardRed1 computes a hash map κ that maps unreduced sequences in the old enumeration $\mathcal{E}$ to unreduced sequences in the new enumeration $\mathcal{E}'$ as follows: Suppose w is an arbitrary unreduced sequence in $\mathcal{E}$. Notice that with forward reduction no additional state will be introduced.

- If $\kappa(w)$ is nil, the state represented by w in $\mathcal{E}$ is eliminated after the change.

- If $\kappa(w)$ is w', the state represented by w in $\mathcal{E}$ is preserved but represented by the unreduced sequence w' (which may or may not be the same as w) in $\mathcal{E}'$.

We start with the empty sequence that represents the initial state, which remains in the modified automaton with the same unreduced sequence λ (Line 2). Then we build more pairs into this mapping in iterations (Lines 4-13). With each iteration we expand the leaf nodes by one level identifying more successor states as being preserved (Lines 5-7), essentially building a spanning tree rooted in the initial state. If a successor state has more than one direct predecessors, all paths are computed and compared to select the first in length-lexicographical order as its corresponding unreduced sequence in the modified enumeration (Lines 8-9). The state previously represented by u in $\mathcal{E}$ is represented by v in $\mathcal{E}'$, and the pair added at a certain iteration (Lines 3, 10-12).

Using this κ mapping we are able to construct the new enumeration $\mathcal{E}'$ from the old one $\mathcal{E}$ line by line. Algorithm *ForwardReduction1* first calls *CompMapForForwardRed1* to compute κ (Lines 1-2). Then it explicitly defines two special sequences in $\mathcal{E}'$: the empty sequence (Line 3) and the sequence v (Line 4). Next it examines each enumerated sequence in $\mathcal{E}$ to see if the transition depicted by this row still exists in the modified automaton (for the same transition to be preserved the start state must be preserved and the transition cannot be redirected). If so a corresponding row in $\mathcal{E}'$ is computed using κ about what new unreduced sequences now represent the two involving states (Lines 5-7). Finally some enumeration entries may need to be added manually (and not converted from old entries) due to a state



(a) Before the forward reduction

(b) After the forward reduction

Figure 2. The Mealy machine before and after forward reducing a to b on $\mathcal{E}_{ex}$

being represented by a different unreduced sequence that has a different legality status after the change (and therefore has to be explicitly extended in $\mathcal{E}'$) (Lines 8-11).

For our example of "forward reducing" Sequence a to Sequence b in $\mathcal{E}_{ex}$, applying Algorithm *CompMapForForwardRed1* gives the following mappings in κ:

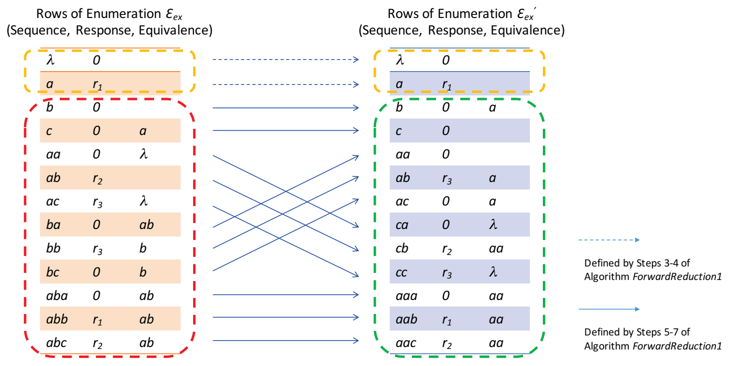
Iteration 0	$\kappa(\lambda) = \lambda$
Iteration 1	$\kappa(a) = c$, $\kappa(b) = a$
Iteration 2	$\kappa(ab) = aa$

Figure 2 shows the automata before and after the change. We label each state with the representing unreduced sequence in the corresponding enumeration. Notice that the arc with the input/output (a/r_1) from the initial state is redirected to point to the same state reached by Sequence b (also from the initial state). Although this is the only arc redirected (all the states are preserved in the modified automaton and all other arcs remain unchanged), every state except the initial state corresponds to a newly computed unreduced sequence as a result of the change.

Except for λ and a which are defined by Steps 3-4 of Algorithm *ForwardReduction1*, all the rows in the modified enumeration $\mathcal{E}'_{ex}$ are computed by Steps 5-7 from $\mathcal{E}_{ex}$ on a line-by-line basis. For instance the original row for Sequence ba being mapped to 0 and reduced to Sequence ab is converted to the following row in $\mathcal{E}'_{ex}$: $\kappa(b)a$ being mapped to 0 and reduced to $\kappa(ab)$, i.e., aa being mapped to 0 and reduced to itself (essentially it is unreduced). Figure 3 shows how each row in $\mathcal{E}'_{ex}$ is constructed from possibly a corresponding row in $\mathcal{E}_{ex}$ by applying our forward reduction algorithm. For this example no new rows need to be added runnning Steps 8-11 of the algorithm.

Case 2 handles the situation when the prior sequence under "forward reduction" is the empty sequence. This affects how transitions are defined out of the initial state. Both algorithms are modified slightly to accommodate this. Because of the page limit we are not including detailed discussion of Case 2 here.

It is worth mentioning that our forward reduction algorithms have polynomial time complexity. They take time


Rows of Enumeration $\mathcal{E}_{ex}$
(Sequence, Response, Equivalence)

λ	0	
a	r_1	
b	0	
c	0	a
aa	0	λ
ab	r_2	
ac	r_3	λ
ba	0	ab
bb	r_3	b
bc	0	b
aba	0	ab
abb	r_1	ab
abc	r_2	ab

Rows of Enumeration $\mathcal{E}_{ex}'$
(Sequence, Response, Equivalence)

λ	0	
a	r_1	
b	0	a
c	0	
aa	0	
ab	r_3	a
ac	0	a
ca	0	λ
cb	r_2	aa
cc	r_3	λ
aaa	0	aa
aab	r_1	aa
aac	r_2	aa

Defined by Steps 3-4 of Algorithm *ForwardReduction1*
Defined by Steps 5-7 of Algorithm *ForwardReduction1*

Figure 3. Constructing $\mathcal{E}'_{ex}$ from $\mathcal{E}_{ex}$ applying Algorithm *ForwardReduction1*

$O(n^2 log(n))$, where n is the size of the enumeration. We hope to emphasize that our forward reduction algorithms (for both cases) were first developed in functional form using an axiom system for sequence-based specification [18] and proved for correctness before they were turned into procedural definitions that facilitate tool implementation. The algorithms were then fully implemented in a newer version of Proto_Seq that is ready for release. Using a theory-based approach ensures that our solution is backed up by sound mathematics and built on a solid theoretical foundation.

4 A case study: The mine pump controller

We did the case study of a mine pump controller software whose requirements are taken from [16]. First we developed two enumerations focusing on different system boundaries. The first enumeration includes all four human inputs from either an ordinary human operator or a special human operator (i.e., the supervisor); both could switch the pump on or off under different conditions. The second enumeration includes all six sensor inputs about the carbon monoxide and the airflow levels, the methane level, detected pump failure, and the water level. Then we merged them into one enumeration that handles all the ten system inputs using a merging procedure we defined for combining specifications. The merged enumeration contains new sequences as placeholders, which were then considered one by one by a human specifier to define the behavior involving the interaction of stimuli from the different smaller specifications. Our forward reduction algorithm was applied in defining these new entries. Table 5 shows the data we collected from the completed three specifications.

The forward reduction algorithm was applied twice in defining new sequences in the merged enumeration:

1. The new sequence **wlh.hpon** corresponds to the following sequence of events: sensor reporting water

Table 5. Data on the enumerations of the mine pump controller software

	First Enumeration	Second Enumeration	Merged Enumeration
# of Stimuli	4	6	10
Termination Enumeration Length	2	5	5
Sequences Extended	2	16	22
Sequences Analyzed	9	129	265

level between low and high limits, then an ordinary human operator switching the pump on. Based on the requirements this gets the system to the same state as a later sequence in length-lexicographical order that has been explored in one of the smaller specifications: **wlh.wah.wlh**. The later sequence corresponds to sensor reporting the water level first between low and high limits, then above the high limit, and then back to between low and high limits again. In this sequence when water level first gets above high the pump controller automatically switches the pump on (same effect as if it were turned on by an ordinary human operator) and it stays on even though water level returns to between low and high, based on the requirements. Without the forward reduction algorithm one would need to extend **wlh.hpon** and its extensions the same way they have treated **wlh.wah.wlh** and its further extensions to explore the state and all its successor states, and all the work they have done on the later sequence would have been erased. With the forward reduction algorithm all the work they have done exploring these states is preserved, and the specification is automatically modified to accommodate all change impacts.

2. As a result of the above forward reduction we have in the merged enumeration **wlh.hpon.l⟨critical()=true⟩** being extended. It represents the following sequence of events: water sensor reporting water level between low and high limits, an ordinary human operator switching the pump on, and then sensor reporting carbon monoxide and airflow levels critical. This gets the system to the same state as an earlier sequence introduced during the merge that has not been explored: **l⟨critical()=true⟩.wlh.hpon**. The forward reduction algorithm was applied again to automatically extend this earlier sequence and make all the needed changes to the specification, eliminating the need for any unnecessary rework.

5 Related work

Sequence-based specification emerged from the functional treatment of software as described by Mills [21, 19, 20]. The development was most directly influenced by the trace-assertion method of Parnas [22, 4] and the algebraic treatment of regular expressions by Brzozowski [8]. Foundations of the sequence-based specification method were established by Prowell and Poore in [24, 26, 23]. An axiom system for sequence-based specification was developed more recently [18] for a formal treatment. The axiom system was used in developing a theory and a set of algorithms that manage requirements changes and state machine changes [17]. The algorithm presented in this paper can be added to that set.

The primary distinction of sequence-based specification from its nearest neighbors (the *trace-assertion method* [11, 9, 10, 15, 22, 4] and *software cost reduction* [12]) is that we evolve the discovery and invent the state machine from requirements through systematic enumeration of input sequences and recording sequence equivalences based on future behavior. Likewise, the theory presented in this paper, as well as that underlying other change algorithms differs from the conventional state change theory in that it is designed for human interactive sequence enumeration and revision, and targeted at an appropriate subset of Mealy machines that can be obtained through enumeration.

6 Conclusion and future work

Sequence-based specification is a rigorous method that converts informal, imprecise descriptions of functional requirements to a precise software specification. The process is iterative in nature due to a human specifier's evolving learning and understanding of the system under specification. Field applications had identified the need for a forward reduction algorithm that could equate an earlier sequence to a later sequence in length-lexicographical order when they result in the same system state. This paper presents an algorithm we have developed for this purpose, using the automaton theory and an axiom system for sequence-based specification. We have fully implemented the algorithm in a supporting tool, which will turn out to be useful and effective not only in deriving a system-level specification but also in merging and combining partial work products towards a formal system model. We illustrate the algorithm with a symbolic example, and report a larger case study from published literature. The enhanced theory, practice, and tool support will facilitate and advance the application of sequence-based specification in field use.

Future work is along the line of scaling sequence-based software specification to very large and complex software-intensive systems. Work is under way to combine, merge, and compose smaller specifications that focus on different system boundaries. We are also conducting a number of other case studies to verify the correctness of the proposed theory and evaluate its effectiveness.

Acknowledgements

The authors would like to thank Tom Swain (previous manager of UTK SQRL) for helping set up an environment for new development and testing, and Xin Guo for earlier code contribution. This work was generously funded by Rockwell Collins, Air Force Research Laboratory, and Ontario Systems through the NSF Security and Software Engineering Research Center (S^{2}ERC).

References

[1] 2010. Jesse H. Poore. Private communication.

[2] 2016. Prototype Sequence Enumeration (Proto_Seq). Software Quality Research Laboratory, The University of Tennessee. https://sourceforge.net/projects/protoseq/.

[3] 2016. Requirements Elicitation and Analysis with Sequence-Based Specification (REALSBS). Software Quality Research Laboratory, The University of Tennessee. http://realsbs.sourceforge.net.

[4] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Proceedings of the 2nd Conference of the European Cooperation on Informatics*, pages 211–236, Venice, Italy, 1978.

[5] T. Bauer, T. Beletski, F. Boehr, R. Eschbach, D. Landmann, and J. Poore. From requirements to statistical testing of embedded systems. In *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*, pages 3–9, Minneapolis, MN, 2007.

[6] L. Bouwmeester, G. H. Broadfoot, and P. J. Hopcroft. Compliance test framework. In *Proceedings of the 2nd Workshop on Model-Based Testing in Practice*, pages 97–106, Enscede, The Netherlands, 2009.

[7] G. H. Broadfoot and P. J. Broadfoot. Academia and industry meet: Some experiences of formal methods in practice. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference*, pages 49–59, Chiang Mai, Thailand, 2003.

[8] J. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[9] J. Brzozowski. Representation of a class of nondeterministic semiautomata by canonical words. *Theoretical Computer Science*, 356:46–57, 2006.

[10] J. Brzozowski and H. Jürgensen. Representation of semiautomata by canonical words and equivalences. *International Journal of Foundations of Computer Science*, 16(5):831–850, 2005.

[11] J. Brzozowski and H. Jurgensen. Representation of semiautomata by canonical words and equivalences, part II: Specification of software modules. *International Journal of Foundations of Computer Science*, 18(5):1065–1087, 2007.

[12] C. L. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley-Interscience, 2001.

[13] P. J. Hopcroft and G. H. Broadfoot. Combining the box structure development method and CSP for software development. *Electronic Notes in Theoretical Computer Science*, 128(6):127–144, 2005.

[14] D. Jackson, M. Thomas, and L. I. Millett, editors. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[15] R. Janicki and E. Sekerinski. Foundations of the trace assertion method of module interface specification. *IEEE Transactions on Software Engineering*, 27(7):577–598, 2001.

[16] M. Joseph, editor. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall International, London, United Kingdom, 1996.

[17] L. Lin, S. J. Prowell, and J. H. Poore. The impact of requirements changes on specifications and state machines. *Software: Practice and Experience*, 39(6):573–610, 2009.

[18] L. Lin, S. J. Prowell, and J. H. Poore. An axiom system for sequence-based specification. *Theoretical Computer Science*, 411(2):360–376, 2010.

[19] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley, 1979.

[20] H. D. Mills. The new math of computer programming. *Communications of the ACM*, 18(1):43–48, 1975.

[21] H. D. Mills. Stepwise refinement and verification in box-structured systems. *IEEE Computer*, 21(6):23–36, 1988.

[22] D. L. Parnas and Y. Wang. The trace assertion method of module interface specification. Technical Report 89-261, Queens University, 1989.

[23] S. J. Prowell and J. H. Poore. Sequence-based software specification of deterministic systems. *Software: Practice and Experience*, 28(3):329–344, 1998.

[24] S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29(5):417–429, 2003.

[25] S. J. Prowell and W. T. Swain. Sequence-based specification of critical software systems. In *Proceedings of the 4th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technology*, Columbus, OH, 2004.

[26] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, Reading, MA, 1999.