Front. Comput. Sci. China 2008, 2(4): 331–343
DOI 10.1007/s11704-008-0034-7

RESEARCH ARTICLE

# Pushing requirements changes through to changes in specifications

**Lan LIN (✉), Jesse H. POORE**

Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996-3450, USA

**Abstract** Requirements changes can occur both during and after a phase of development for a variety of reasons, including error correction and feature changes. It is difficult and intensive work to integrate requirements changes made after specification is completed. Sequence-based specification was developed to convert ordinary functional software requirements into complete, consistent, and traceably correct specifications through a constructive process. Algorithms for managing requirements changes meet a very great need in field application of the sequence-based specification method. In this paper we propose to capture requirements changes as a series of atomic changes in specifications, and present polynomial-time algorithms for managing these changes. The algorithms are built into the tool support with which users are able to push requirements changes through to changes in specifications, maintain old specifications over time and evolve them into new specifications with the least amount of human interaction and rework. All our change algorithms are supported by rigorous mathematical formulation and proof of correctness. The application example is a safe controller.

**Keywords** requirements change management, sequence-based software specification, changing state machine diagrams, Mealy machine, automaton

## 1 Introduction

The method of *sequence-based specification* [1,2] was developed to convert ordinary functional requirements into rigorous black box and state box [3,4] specifications through a constructive process. This process is called *sequence enumeration* and assisted with a prototype tool (Proto_Seq) developed by the UTK SQRL lab[*]. The strength of sequence-based specification is in the constructive process it provides for discovering and deriving a state machine of the system (a state box) from the informal requirements, which can be used to greatly facilitate the generation of code, or at least code structure (a clear box) [5]. Sequence-based specification together with the Proto_Seq tool has been effectively used in a variety of real applications, ranging from automotive components to medical devices to scientific instrumentation [5,6].

Enumeration clarifies requirements by noticing any omissions, inconsistencies, and errors inherent in the informal requirements and actively resolving these problems. The result is a specification that is *complete*, *consistent*, and *traceably correct*. This means for any stimulus (input) sequence its mapped response (output) by the black box function is uniquely determined by the formal specification – the completed enumeration, and all the decisions made in deriving the specification can be traced back to requirements.

However, the correct enumeration is not obtained in one pass due to our limited, evolving understanding of the system under specification. We start over several times as the requirements are clarified and corrected. In each iteration we work on a specification obtained from the previous iteration and make changes according to the improved understanding or interpretation of requirements. These successive elemental changes altogether transform one complete and consistent specification into another.

Requirements also change after a phase of development, driven by market and customer demand. Changes in requirements necessitate a series of atomic changes in specifications and code that need to be identified and accommodated.

It is difficult and intensive work to integrate requirements changes made after specification is completed. In either case we need to push requirements changes through

Received July 19, 2008; accepted September 25, 2008

E-mail: {lin, poore}@eecs.utk.edu

[*] Software Quality Research Lab, Department of Electrical Engineering and Computer Science, University of Tennessee at Knoxville, http://www.cs.utk.edu/sqrl/

to changes in specifications. We need algorithms for managing all possible requirements changes and handling the change consequences gracefully. Since sequence-based specifications are developed with the Proto_Seq enumeration tool, our goal is to maximize the automation by the tool and minimize the amount of human interaction and effort involved in changing the specifications in response to changes in requirements. With the automated tool support old specifications can be maintained and evolved into new specifications. A substantial amount of rework is avoided.

This paper is organized as follows. The next section gives an overview of the sequence-based specification method and links to its corresponding state machines. Section 3 introduces all possible atomic requirements changes, but only picks a few for presentation and detailed explanation due to the page limit. It also gives the time complexity analysis of all the change algorithms. Section 4 applies our change theory to a safe controller example. Section 5 compares our approach with similar work in literature and concludes this paper with directions for future research.

## 2 Sequence-based specification and its state machines

Sequence-based specification offers a systematic approach to discover and write a formal behavioral description of a software-intensive system. Starting at some level of abstraction, users of the method first identify all possible *stimuli* (inputs) the system may receive and all possible *responses* (outputs) the system may generate, and then enumerate all possible sequences of stimuli in length-lexicographical order, which represent all scenarios of use.

As the name suggests, sequence enumeration is the literal enumeration of sequences of inputs and the assignment of correct outputs to each enumerated sequence. The unique output (which could be an ensemble of outputs at a finer granularity) is the intended response to the most recent stimulus given the sequence of stimulus histories.

If a sequence generates no externally observable behavior we assign a *null* response to it, indicated by the symbol 0. If a sequence is simply impossible to happen we assign an *illegal* response to it, indicated by the symbol $\omega$. Every enumerated sequence is therefore mapped to either an observable response, or 0, or $\omega$. A sequence is called *illegal* when it maps to $\omega$, otherwise, it is called *legal*.

With a finite number of stimuli typical for any real-world application, there is an infinite number of stimulus sequences (of finite length); enumerating all of them is infeasible. Since sequences can be grouped into finitely many equivalence classes based on future behavior (representing finitely many distinct system states), only a finite number of sequences need to be enumerated to explore the system's behavior.

Two sequences are *Mealy equivalent* if and only if they always generate the same response when extended by the same non-empty input sequence. This means two Mealy equivalent sequences need not be mapped to the same response, but their responses with respect to future extensions must always agree. If a sequence is not found Mealy equivalent to any previously enumerated sequence, it is *unreduced*, otherwise, it is *reduced* to the previously enumerated (Mealy equivalent) sequence that is itself unreduced.

While assigning a response to any enumerated sequence, we also check for the possible reduction to a prior sequence under Mealy equivalence. Reducing a sequence to some prior sequence automatically eliminates the need to enumerate any extension of the current sequence, as the behavior of the extension is fully defined by the same extension of the prior sequence due to Mealy equivalence.

If a sequence is mapped to $\omega$, there is no need to enumerate any of its extensions because the extensions must also be impossible to happen (i.e., physically unrealizable).

The enumeration then proceeds as follows. We start with the first sequence in length-lexicographical order – the *empty sequence* of length zero (denoted by $\lambda$) that contains no stimuli. It is mapped to 0 but not reduced as there is no prior sequence to reduce it to. We extend it by every stimulus and get all sequences of length one, which are considered in lexicographical order for responses and possible reductions. Next we only enumerate sequences of length two that are one-symbol extensions of both legal and unreduced sequences of length one, and consider them in lexicographical order for responses and possible reductions. The process continues until all enumerated sequences of a certain length are either mapped to $\omega$ or reduced to prior sequences. From then on no sequence needs to be extended, hence no more is enumerated. We declare the enumeration *complete*. The rules that guide the enumeration process can be expressed in the form of enumeration axioms [7].

In practice we tag the requirements to ease the work of verifying the correctness of every decision made in the specification process. Enumerations are produced in a table. The (finitely many) rows are for all enumerated sequences in length-lexicographical order. The columns are for the mapped responses, reductions, and requirements traces (tags). An enumeration is *complete and finite* if and only if every legal and unreduced sequence has been extended by every (single) stimulus in the table, and of course the table contains only finitely many rows.

The following example of a safe controller is based on [2]. We tag the requirements in Table 1. Note that the last two rows record two derived requirements (D1 and D2)

**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. |
| D1 | Sequences with stimuli prior to system initialization are illegal by system definition. |
| D2 | Re-initialization (power-on) makes previous history irrelevant. |

that were not in the original requirements and only identified during the enumeration process.

The system boundary is cut between the system and a list of interfaces in the environment representing the external power, keypad, door sensor, and lock actuator. We identify all stimuli and responses from the interfaces in Tables 2 and 3. To make work efficient we use $G$ to denote entering the correct three digits in order and $B$ to denote entering the combination incorrectly but up to the first mistake. A complete enumeration for the safe is constructed in Table 4.

**Table 2** Safe controller stimuli

| Stimulus | Description | Interface |
|---|---|---|
| *B* | Bad digits | keypad |
| *C* | Clear key press | keypad |
| *D* | Door closed | door sensor |
| *G* | Good digits | keypad |
| *L* | Power on with door locked | power, door sensor |
| *U* | Power on with door unlocked | power, door sensor |

**Table 3** Safe controller responses

| Response | Description | Interface |
|---|---|---|
| *lock* | Locking the door | lock actuator |
| *unlock* | Unlocking the door | lock actuator |

For reasons of practicality we do not consider reductions for illegal sequences (as they cannot happen). We can simply treat the first illegal sequence (sequence $B$) as unreduced and all other illegal sequences as reduced to the first illegal sequence. In our theoretical and formal treatment, reductions can happen between legal and illegal sequences as long as Mealy equivalence holds. Our change algorithms are based on the general model.

The enumeration in Table 4 can be read as follows. First, the empty sequence represents the initial system state (no input has been received). It is mapped to 0 and unreduced as enforced by the rules of the method. Then it is extended by every stimulus to get all the sequences of length one (sequence $B$ through sequence $U$).

**Table 4** Safe controller sequence enumeration

| Sequence | Response | Equivalence | Trace |
|---|---|---|---|
| $\lambda$ | 0 | | Method |
| *B* | $\omega$ | | D1 |
| *C* | $\omega$ | | D1 |
| *D* | $\omega$ | | D1 |
| *G* | $\omega$ | | D1 |
| *L* | 0 | | 5 |
| *U* | 0 | | 5 |
| *LB* | 0 | | 1,2,7 |
| *LC* | 0 | *L* | 2 |
| *LD* | $\omega$ | | 8 |
| *LG* | *unlock* | *U* | 1,3 |
| *LL* | 0 | *L* | 5,D2 |
| *LU* | 0 | *U* | 5,D2 |
| *UB* | 0 | *U* | 6 |
| *UC* | 0 | *U* | 6 |
| *UD* | *lock* | *L* | 4 |
| *UG* | 0 | *U* | 6 |
| *UL* | 0 | *L* | 5,D2 |
| *UU* | 0 | *U* | 5,D2 |
| *LBB* | 0 | *LB* | 2 |
| *LBC* | 0 | *L* | 2 |
| *LBD* | $\omega$ | | 8 |
| *LBG* | 0 | *LB* | 2 |
| *LBL* | 0 | *L* | 5,D2 |
| *LBU* | 0 | *U* | 5,D2 |

Among the length-one sequences only two of them, namely $L$ and $U$, are legal, as any event prior to the power-on event cannot be perceived by software by derived requirement D1. Since sequence $B$ is the first illegal sequence in length-lexicographical order, it is unreduced; later illegal sequences are all reduced to $B$. Sequences $L$ and $U$ represent two distinct states of the system other than the initial state (power is turned on but the door can be either locked or unlocked). They both map to 0 as there is no externally observable behavior when power is turned on, and they get extended by every stimulus to form all the enumerated sequences of length two (sequence $LB$ through sequence $UU$).

The length-two sequences are considered in lexicographical order (the order they show up in the table) for the generated responses and possible reductions to prior sequences. For instance, sequence $LG$ represents the usage scenario where power is first turned on with the door of the safe being locked followed by a good combination entry. This will unlock the door and hence the response of $LG$ is mapped to *unlock*. Sequence $LG$ gets the system to the same state as the prior sequence $U$, as in both cases the system has power and the door is unlocked. The decisions regarding the response and the equivalence of $LG$ are traced to requirements 1 and 3.

Similarly the only legal and unreduced sequence of length two, sequence $LB$, is extended to get all enumerated sequences of length three. There are six of them (sequence $LBB$ through sequence $LBU$) and it turns out that all of them are either illegal or reduced to prior sequences, therefore, the enumeration terminates at length three.

A complete and finite enumeration encodes a finite state automaton with Mealy outputs (a Mealy machine [8]). The set of states corresponds to the set of equivalence classes represented by all unreduced sequences. Transitions between states are dictated by reductions. Outputs on the arcs are dictated by response mappings. The enumeration rules put more constraints on the structure of the Mealy machines derived from enumerations. We call this special subset of Mealy machines *enumeration Mealy machines*. A few conditions are enforced in addition to the enumeration rules to establish a one-to-one correspondence from all complete and finite enumerations onto all enumeration Mealy machines (see [7] for a rigorous and formal treatment). Informally we describe the characteristics of an enumeration Mealy machine as follows:

1. For an $n$-state enumeration Mealy machine, the set of states are named $q_0, \ldots, q_{n-1}$;
2. Every state is reachable (connected) from the initial state $q_0$;
3. Compute the first word $c_i$ in length-lexicographical order that takes the automaton from the initial state $q_0$ to state $q_i$, then the sequence of words $c_0, \ldots, c_{n-1}$ are in length-lexicographical order;
4. Extend the output function following [8] to all input sequences: the output for the empty word is 0; the output for any non-empty word is the output of the last transition on the path generated by the word from the initial state. If $c_i$ as defined above has $\omega$ as its output, then state $q_i$ is a trap state (i.e., all outgoing arcs from $q_i$ return to $q_i$);
5. If one incoming arc to state $q_i$ has $\omega$ as its output, then all outgoing arcs from $q_i$ have $\omega$ as their output.

Our enumeration in Table 4 encodes the Mealy machine diagrammed in Fig. 1, with states $q_0$–$q_4$ represented by unreduced sequences $\lambda$, $B$, $L$, $U$, and $LB$ from the enumeration, respectively.

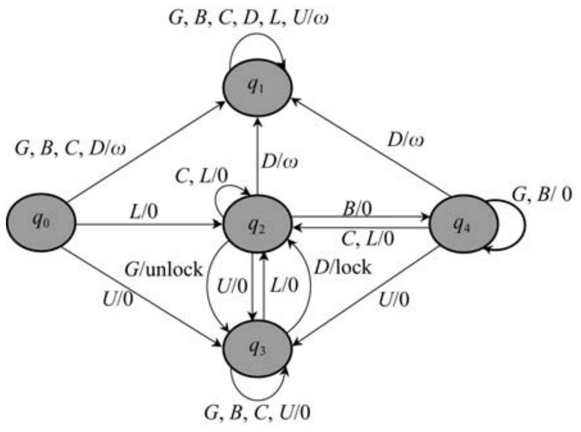


**Fig. 1** A state machine for the safe controller (states $q_0$–$q_4$ represented by unreduced sequences $\lambda$, $B$, $L$, $U$, and $LB$ respectively)

There is an important connection between enumerations and enumeration Mealy machines. Given any enumeration Mealy machine, if we compute for every state the first word in length-lexicographical order that reaches the state from the initial state, then the set of computed words are exactly the set of all unreduced sequences in the corresponding enumeration. This fact has significant implication on requirements changes. The state machine is changed as a result of changes in requirements. Change effect on state machines is more intuitively seen than change effect on specifications (enumerations). This hints at using state machines to model and manage requirements changes and specification changes. Through recomputing the special set of words associated with states of the modified automaton, we are able to obtain all unreduced sequences in the modified enumeration, and construct the new enumeration accordingly. This strategy is used frequently in algorithms for handling requirements changes with complicated rippling effect (i.e., a single change resulting in a considerable part of the enumeration being changed).

A complete and finite enumeration uncovers the *black box function* of the system that maps every stimulus sequence to a response. An algorithm for its computation (Algorithm *BlackBox*) [2] mimics deriving the output for the given input sequence by the extended output function of the corresponding Mealy machine.

```
Algorithm BlackBox($\mathcal{E}$, $u$)
Input: A complete and finite enumeration $\mathcal{E}$, a stimulus
       sequence $u$
Output: The response for $u$
(* Compute the black box function*)
1. while true
2.    do if $u$ is enumerated in $\mathcal{E}$
3.          then return the mapped response of $u$
4.       Find the longest prefix $u'$ of $u$ that is enumerated
         in $\mathcal{E}$.
```

```
5.  if $u'$ is illegal
6.      then return $\omega$
7.      else Let $u'$ be reduced to $v$ in $\mathcal{E}$. Replace $u'$ with $v$
            in $u$ to form a new sequence $u''$.
8.          return $BlackBox(\mathcal{E}, u'')$
```

As an example, we show how to derive the correct response to an arbitrary stimulus sequence, say *LBCUDG*, by the available algorithm. Since *LBC* is the longest prefix of *LBCUDG* that gets enumerated in Table 4, and *LBC* is a legal sequence reduced to prior sequence *L*, we replace *LBC* with *L* and run the algorithm on the newly formed sequence *LUDG*. Since the longest prefix of *LUDG* that gets enumerated is *LU*, which is legal and reduced to *U*, we run the algorithm on *UDG*. Then *UD* is replaced with *L*, and finally the newly formed sequence *LG* maps to *unlock* by Table 4.

## 3 Managing atomic requirements changes

We propose to capture requirements changes as a series of atomic changes in specifications, and present algorithms for managing all possible atomic requirements changes to a sequence-based specification. The algorithms address atomic changes one at a time, in order to preserve completeness, consistency, and correctness. As may be noticed by the reader, seemingly minor changes in requirements may grow into long lists of atomic changes for the specification and the code.

Integrating requirements changes into the specification iteratively requires a human specifier identify the next atomic change needed. Automation is built into the Proto_Seq enumeration tool to manage each atomic change with an algorithm that handles all its consequences and produces an evolving specification. From our experience, these change algorithms are needed and used extensively for almost any application for which sequence-based specification is used, due to the fact that the correct specification of the system is never built in one pass but rather an evolving product of the first few unsuccessful trials.

Table 5 lists all possible atomic requirements changes to a sequence-based specification. Firstly, the stimulus set could be changed as we identify a new stimulus across the system boundary or an old one no longer of interest. Possible *stimulus changes* include adding a stimulus, deleting a stimulus, and their combinations thereof. Secondly, the response mapping could be changed for any enumerated sequence. The new response could be different from any of the old responses and emerge from the new or changed requirements. A *response change* refers to changing the mapped response of a specific enumerated sequence in a complete and finite enumeration and handling all its consequences. Depending on the legality of this sequence before and after the change, the response change is classified as from legal to legal, legal to illegal, or illegal to legal; the latter two cases are also considered as *legality changes*. Lastly, the declared reduction could be changed for any enumerated sequence. An *equivalence change* refers to changing the reduction of a specific enumerated sequence in a complete and finite enumeration and handling all its consequences. Depending on the status of the sequence before and after the change (reduced or unreduced, legal or illegal), again there are subtleties.

The reader may note that we do not have additions or deletions for responses as for stimuli; these are handled by the elemental response change operation. When a new response is introduced, we are actually mapping at least one enumerated sequence to this new response. When a response is deleted, we are also redefining all sequences previously mapped to this response. Therefore, the response change operation covers these situations sufficiently.

The algorithms were first formulated and described as functions [7] using an axiom system we have developed for a rigorous treatment of the sequence-based specification method. They were then proven for correctness and fully implemented in the Proto_Seq enumeration tool. Here we will present them informally with an attempt to highlight the consequences of changes and motivate the decisions made. Because of the page limitation we will pick a few algorithms for detailed explanation, and refer the reader to [7,9] for the rest of the algorithms. Some chosen

**Table 5** Summary of all possible atomic requirements changes

| | | | |
|---|---|---|---|
| stimulus changes | adding a stimulus | | Algorithm 1 |
| | deleting a stimulus | | Algorithm 2 |
| response changes | from legal to legal | | Algorithm 3 |
| | from illegal to legal | for a reduced sequence | Algorithm 4 |
| | | for an unreduced sequence | Algorithm 5 |
| | from legal to illegal | for a reduced sequence | Algorithm 6 |
| | | for an unreduced sequence | Algorithm 7 |
| equivalence changes | for an unreduced illegal sequence | | Algorithm 8 |
| | for an unreduced legal sequence | | Algorithm 9 |
| | for a reduced illegal sequence | | Algorithm 10 |
| | for a reduced legal sequence | keeping it reduced | Algorithm 11 |
| | | making it unreduced | Algorithm 12 |

algorithms have the most comprehensive or far-reaching impact of changes.

In any case we start with a complete and finite enumeration $\mathcal{E}$, perform all required changes incurred by a single atomic change. It is proven in [7] that the resulting enumeration $\mathcal{E}'$ guarantees to be also complete and finite.

## 3.1 Adding a stimulus: Algorithm 1

Suppose a new stimulus $x$ is added into the set of stimuli. Since all legal and unreduced sequences in $\mathcal{E}$ must have been extended by every old stimulus, their extensions by $x$ need to be defined as well. To minimize human intervention, the tool makes an assumption about all new extensions by mapping them to $\omega$. It is up to the user of the tool to decide later what the correct response and reduction will be according to the new requirements. As a reminder to the user, all new extensions that are mapped to $\omega$ are highlighted in the enumeration table. The user can redefine these entries easily by calling response or equivalence change algorithms.

To summarize, adding a stimulus has the following implications:

- All old enumeration entries remain unchanged;
- All legal and unreduced sequences in $\mathcal{E}$ are extended by $x$. The extensions are mapped to $\omega$ and highlighted.

```
Algorithm 1 (AddStim) ($\mathcal{E}$, $x$)
Input: A complete and finite enumeration $\mathcal{E}$, a new
       stimulus $x$
Output: A complete and finite enumeration $\mathcal{E}'$
(*Add a stimulus*)
1. Initialize $\mathcal{E}'$ to be the same as $\mathcal{E}$.
2. for every legal and unreduced sequence $u$ in $\mathcal{E}'$
3.     do Extend $u$ by $x$ in $\mathcal{E}'$. Map the extension to $\omega$ and
          make it unreduced.
4.        Highlight the row for sequence $ux$.
5. return $\mathcal{E}'$
```

## 3.2 Deleting a stimulus: Algorithm 2

Suppose stimulus $x$ is to be removed from the set of stimuli. In the view of the corresponding Mealy machine all arcs labeled with $x$ will disappear. Some states may become unreachable as a result and have to be removed.

For any of the remaining states, say $q_i$, if previously the first word in length-lexicographical order reaching it from the initial state was $c_i$, and $c_i$ contains the bad stimulus $x$, then it no longer exists as a path in the modified automaton, hence the word for $q_i$ in the new automaton needs to be recomputed. We remain interested in the first words in length-lexicographical order reaching every state of the modified automaton from the initial state because they are exactly unreduced sequences in the modified enumeration.

There are more subtleties. Now suppose the newly computed word for $q_i$ is $c_i'$, which does not contain $x$ as a symbol. If the output for $c_i$ was not $\omega$, but the output for $c_i'$ is $\omega$, and $q_i$ was not previously a trap state, then all outgoing arcs from $q_i$ need to be redirected in the modified automaton to make it a trap state, in accordance with Condition 4 required for any enumeration Mealy machine (see Section 2). This may render more states unreachable and getting removed. While recomputing the special words for every state of the new automaton (that correspond to new unreduced sequences), we need to be careful with these redirected arcs and newly introduced trap states.

As an example Fig. 2(a) shows the state machine for an enumeration with stimuli $a$ and $b$, and Fig. 2(b) shows its modified version after $a$ is deleted. To visualize the relationship of enumerations and state machines we label each state with its corresponding unreduced sequence in the old (or new) enumeration. Note that the old state $a$ is preserved but named after $bb$, and since $bb$ outputs $\omega$ it becomes a trap state and further eliminates the old state $aa$.

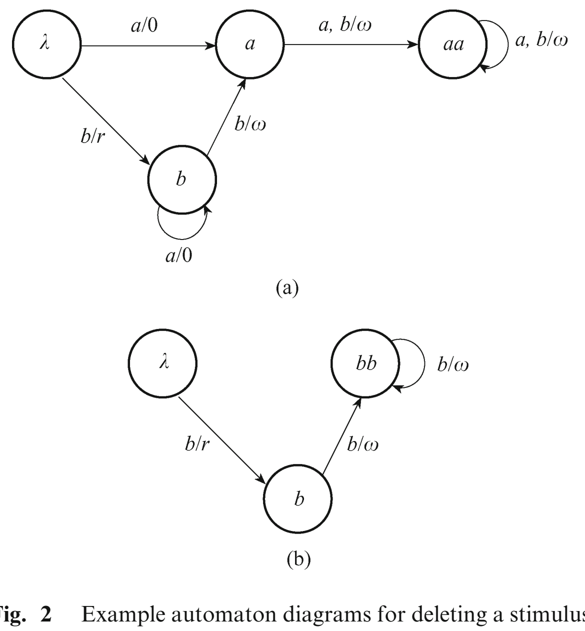


**Fig. 2** Example automaton diagrams for deleting a stimulus (a) before deleting $a$, (b) after deleting $a$

Once the $c_i'$s are computed, we are ready to derive the new enumeration from the old one mechanically, by linking old enumeration entries to old Mealy machine transitions (with outputs on the arcs), and linking new Mealy machine transitions to new enumeration entries. In general, if the transition from state $q_i$ to state $q_j$ by arc $s$ with output $r$ is preserved in the modified automaton, then we have:

- In $\mathcal{E}$ unreduced sequence $c_i$ when extended by stimulus $s$ is mapped to response $r$ and reduced to unreduced sequence $c_j$;
- In $\mathcal{E}'$ unreduced sequence $c_i'$ when extended by stimulus $s$ is mapped to response $r$ and reduced to unreduced sequence $c_j'$.

For example the transition from state $b$ on input $b$ corresponds to "$bb$ mapped to $\omega$ and reduced to $a$" in the old enumeration, but "$bb$ mapped to $\omega$ and reduced to $bb$" in the new one. From here on, we will refer to an unreduced sequence interchangeably as "reduced to itself".

Exceptions to this general rule exist. In case $c_i$ has $\omega$ as its output, the transition does not have a corresponding entry in $\mathcal{E}$ as illegal sequences are not extended, but it is implied. However, if the output for $c_i'$ is not $\omega$, then the transition indeed has a corresponding entry in $\mathcal{E}'$, which cannot be obtained by revising an existing old entry in $\mathcal{E}$ and must be added explicitly.

Of course we need to know when a transition in the old automaton, say from state $q_i$ to state $q_j$, is preserved in the new automaton to apply the substitution rule. This is easy given the helper function that computes the (partial) mapping from all unreduced sequences in $\mathcal{E}$ to all unreduced sequences in $\mathcal{E}'$. The transition from $q_i$ to $q_j$ is preserved in the new automaton in all but the following two cases:

- State $q_i$ becomes unreachable after deleting $x$, hence there is not a primed value for $c_i$;
- State $q_i$ is not a trap state in $\mathcal{E}$, however, $c_i'$ has output $\omega$ which enforces $q_i$ to become a trap state in $\mathcal{E}'$. The redirected transitions do not have any entries in $\mathcal{E}'$ as illegal sequences are not extended.

To summarize, deleting a stimulus has the following implications:

- Unreduced sequences in $\mathcal{E}$ that do not contain $x$ remain as unreduced sequences in $\mathcal{E}'$;
- Unreduced sequences in $\mathcal{E}$ that contain $x$ may or may not have corresponding unreduced sequences in $\mathcal{E}'$. In case they have, the unreduced sequences in $\mathcal{E}'$ need to be computed;
- $\mathcal{E}'$ can be derived from $\mathcal{E}$ by applying substitution rules and covering a few exceptional cases.

Algorithm *CompMapForStimDel* below serves as a helper function that computes a partial mapping $\kappa$ from all unreduced sequences in $\mathcal{E}$ to all unreduced sequences in $\mathcal{E}'$. Given an unreduced sequence $u$ in $\mathcal{E}$, if $u$ does not contain $x$ as a symbol, then it remains as an unreduced sequence in $\mathcal{E}'$ representing the same state (Steps 2-4). These states provide the basis for computing which other old states are also preserved by the following observation. Every enumerated sequence in $\mathcal{E}$ except the empty sequence describes a transition of the old automaton. For instance, if prefix sequence $p$ followed by stimulus $s$ is reduced to sequence $v$, then the state represented by $p$ has an outgoing arc $s$ into the state represented by $v$. If the starting state but not the ending state of this transition is known to be existing in the modified automaton ($\kappa(p) \neq$ **nil**, $\kappa(v) =$ **nil**), the arc is not to be deleted ($s \neq x$), and the arc is not to be redirected ($BlackBox(\mathcal{E}, \kappa(p)) \neq \omega$), then the ending state must be preserved in the new automaton and reached by $\kappa(p)$ concatenated with $s$ (Steps 6-8). All incoming arcs to such states are considered to select the first paths in length-lexicographical order getting to these states as their corresponding new unreduced sequences in $\mathcal{E}'$ (Steps 9-10).

```
Algorithm CompMapForStimDel($\mathcal{E}$, $x$)
Input: A complete and finite enumeration $\mathcal{E}$, an existing
       stimulus $x$
Output: A hash map $\kappa$ 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 deleting $x$), or nil (otherwise)
(* Compute the map for unreduced sequences when *)
(* deleting a stimulus *)
1. Initialize an empty hash map $\kappa$.
2. for each unreduced sequence $u$ in $\mathcal{E}$
3.     do if $u$ does not contain $x$ as a symbol
4.         then $\kappa(u) \leftarrow u$
5. repeat
6.     for every enumerated sequence in $\mathcal{E}$ of the form:
       prefix sequence $p$ followed by stimulus $s$ is reduced
       to $v$
7.         do if $s \neq x$ and $\kappa(v) =$ nil and $\kappa(p) \neq$ nil and
           $BlackBox(\mathcal{E}, \kappa(p)) \neq \omega$
8.             then Let $\kappa(p)$ concatenated with $s$ be a candid-
                    ate for $\kappa(v)$.
9.     for every unreduced sequence $v$ that has designated
       candidates in Steps 6-8
10.        do $\kappa(v) \leftarrow$ its first candidate in length-lexico-
              graphical order
11. until The last iteration has no new sequence defined
       for $\kappa$.
12. return $\kappa$
```

The stimulus deletion algorithm (Algorithm 2 (*DelStim*)) first calls Algorithm *CompMapForStimDel* to get the mapping $\kappa$ from old unreduced sequences to new unreduced sequences (Steps 1-2). Then it initializes $\mathcal{E}'$ with the empty sequence (Step 3). Next, every enumerated non-empty sequence (prefix sequence $p$ concatenated with stimulus $s$) in $\mathcal{E}$ is considered. If the corresponding transition is neither deleted ($s \neq x$ and $\kappa(p) \neq$ **nil**) nor redirected ($BlackBox(\mathcal{E},\kappa(p)) \neq \omega$), then the preserved transition translates to an enumerated sequence in $\mathcal{E}'$ (Steps 4-6). Finally, every old trap state corresponding to an illegal and unreduced sequence $u$ in $\mathcal{E}$ is examined. If the same state is preserved ($\kappa(u) \neq$ **nil**) but represented by a legal and unreduced sequence $\kappa(u)$ in $\mathcal{E}'$ ($BlackBox(\mathcal{E},\kappa(u)) \neq \omega$), then $\kappa(u)$ is explicitly extended (Steps 7-10).

```
Algorithm 2 (DelStim)($\mathcal{E}$, $x$)
Input: A complete and finite enumeration $\mathcal{E}$, an existing
       stimulus $x$
Output: A complete and finite enumeration $\mathcal{E}'$
(* Delete a stimulus *)
1. Initialize an empty hash map $\kappa$.
2. $\kappa \leftarrow CompMapForStimDel(\mathcal{E}, x)$
3. Initialize $\mathcal{E}'$ to contain $\lambda$ only, with $\lambda$ mapped to 0 and
   unreduced.
4. 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 $v$
5.    do if $s \neq x$ and $\kappa(p) \neq$ nil and $BlackBox(\mathcal{E}, \kappa(p)) \neq \omega$
6.       then Add the following sequence in $\mathcal{E}'$: prefix sequence $\kappa(p)$ followed by stimulus $s$ mapped to response $r$ and reduced to sequence $\kappa(v)$.
7. for every enumerated illegal and unreduced sequence $u$ in $\mathcal{E}$
8.    do if $\kappa(u) \neq$ nil and $BlackBox(\mathcal{E}, \kappa(u)) \neq \omega$
9.       then for every stimulus $s$ except $x$
10.         do Add the following sequence in $\mathcal{E}'$: prefix sequence $\kappa(u)$ followed by stimulus $s$ mapped to $\omega$ and reduced to sequence $\kappa(u)$.
11. return $\mathcal{E}'$
```

It is proven [7] that combinations of stimulus addition and deletion can be performed in an arbitrary order without affecting the final result.

## 3.3 Changing a response from legal to illegal for an unreduced sequence: Algorithm 7

Suppose we pick an enumerated legal, unreduced sequence $u$ and want to change its response to $\omega$. All extensions of $u$ will become illegal after the change. If there is any sequence, say $w$, previously reduced to $u$, keeping the same reduction may pose a problem as we may not want to change the response for any extension of $w$ (let alone to $\omega$).

This problem is caused by assuming $u$ and $w$ are still taking the automaton to the same state after the change but actually they are not. This happens only if $w$ is not an extension of $u$. The state reachable by both in the old automaton is preserved in the new automaton by $w$ (and possibly many other sequences). Sequence $u$ leads to a newly added trap state that defines the new behavior for $u$ as well as any extension of $u$. The state that gets preserved now corresponds to a different unreduced sequence than $u$ in $\mathcal{E}'$, which must be the first word in length-lexicographical order taking the modified automaton from the initial state to the preserved state.

The same problem exists for reductions in $\mathcal{E}$ to an unreduced sequence in which $u$ is a proper prefix. The state, even if preserved, may correspond to a different unreduced sequence in $\mathcal{E}'$ that cannot contain $u$ as a prefix.

Therefore, the unreduced sequences in $\mathcal{E}$ that contain $u$ as a prefix (not necessarily proper) are problematic. The states represented by them in the old automaton may not be preserved in the new automaton (in that case all sequences getting to these states get to the newly added trap state in the new automaton), or they are preserved but represent different unreduced sequences in $\mathcal{E}'$. We have a helper function similar to the one used for stimulus deletion to compute which of these states get preserved as well as their corresponding unreduced sequences in $\mathcal{E}'$. Again we need to handle subtleties resulted from trying to satisfy Condition 4 for an enumeration Mealy machine (see Section 2).

As an example Fig. 3 shows the state machines before and after changing the response of a legal, unreduced sequence $a$ from 0 to $\omega$.

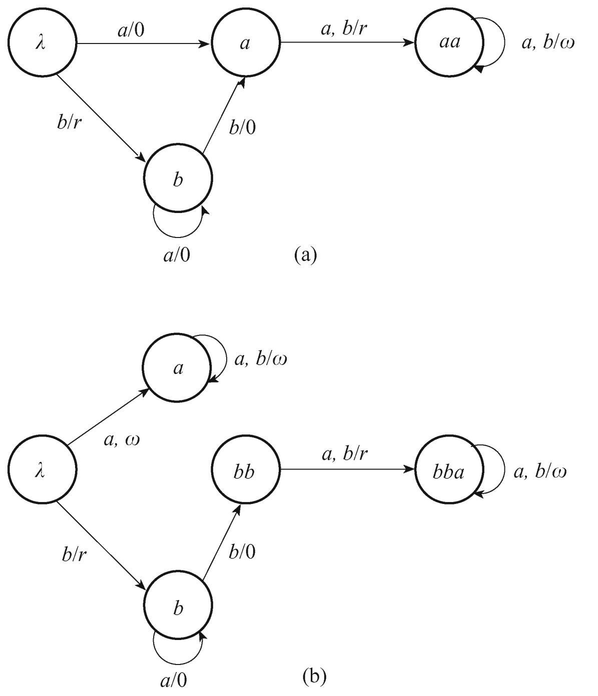


**Fig. 3** Example automaton diagrams for changing a response from legal to illegal for an unreduced sequence (a) before changing the response of $a$, (b) after changing the response of $a$

Once we have all unreduced sequences in $\mathcal{E}'$ computed, we can derive $\mathcal{E}'$ from $\mathcal{E}$ in a similar fashion as for stimulus deletion. Since $u$ leads to a newly added trap state, the row for $u$ in the old enumeration table is not used in substitution. The row for $u$ is defined explicitly for $\mathcal{E}'$.

To summarize, changing a response from legal to illegal for an unreduced sequence has the following implications:

- Unreduced sequences in $\mathcal{E}$ that do not contain $u$ as a prefix remain as unreduced sequences in $\mathcal{E}'$;
- Unreduced sequences in $\mathcal{E}$ that contain $u$ as a prefix may or may not have corresponding unreduced sequences in $\mathcal{E}'$. In case they have, the unreduced sequences in $\mathcal{E}'$ need to be computed;
- Sequence $u$ is unreduced in $\mathcal{E}'$, but not related to any unreduced sequence in $\mathcal{E}$. It is defined explicitly;
- $\mathcal{E}'$ can be derived from $\mathcal{E}$ by applying substitution rules and covering a few exceptional cases.

Algorithm *CompMapForRespOrEquivChg* serves as a helper function for both the response change algorithm discussed in this section (Algorithm 7 (*ChgResp5*)) and the equivalence change algorithm discussed in the next section (Section 4, Algorithm 9 (*ChgEquiv2*)). In both cases a formerly extended sequence will not be extended after being mapped to $\omega$ or reduced to a prior sequence. Algorithm *CompMapForRespOrEquivChg* computes the mapping from unreduced sequences in $\mathcal{E}$ to unreduced sequences in $\mathcal{E}'$. As a starting point, if an unreduced sequence in $\mathcal{E}$ does not contain the sequence $u$ under change as a prefix, then it remains as an unreduced sequence in $\mathcal{E}'$ representing the same state (Steps 2-4). Similar as for Algorithm *CompMapForStimDel*, every enumerated non-empty sequence in $\mathcal{E}$ describes a transition in the old automaton. For instance, if prefix sequence $p$ followed by stimulus $s$ is reduced to sequence $v$, then the state represented by $p$ has an outgoing arc $s$ into the state represented by $v$. If the starting state but not the ending state is known to be existing in the modified automaton ($\kappa(p) \neq$ **nil**, $\kappa(v) =$ **nil**), and the transition is not to be redirected ($ps \neq u$, $BlackBox(\mathcal{E},\kappa(p)) \neq \omega$), then the state represented by $v$ in the old automaton is preserved in the new automaton and reached by $\kappa(p)$ concatenated with $s$ (Steps 6-8). All incoming arcs to such states are considered to pick the first paths in length-lexicographical order leading to these states as the corresponding new unreduced sequences in $\mathcal{E}'$ (Steps 9-10).

```
Algorithm CompMapForRespOrEquivChg($\mathcal{E}$, $u$)
Input: A complete and finite enumeration $\mathcal{E}$, an enumerated legal, unreduced sequence $u$ except $\lambda$
Output: A hash map $\kappa$ 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 mapping $u$ to $\omega$, or reducing $u$ to a prior sequence), or nil (otherwise)
(* Compute the map for unreduced sequences when *)
(* changing a formerly extended sequence *)
1. Initialize an empty hash map $\kappa$.
2. for each unreduced sequence $v$ in $\mathcal{E}$
3.     do if $v$ does not contain $u$ as a prefix
4.         then $\kappa(v) \leftarrow v$
5. repeat
6.     for every enumerated sequence in $\mathcal{E}$ of the form: prefix sequence $p$ followed by stimulus $s$ is reduced to $v$
7.         do if $ps \neq u$ and $\kappa(v) =$ nil and $\kappa(p) \neq$ nil and $BlackBox(\mathcal{E}, \kappa(p)) \neq \omega$
8.             then Let $\kappa(p)$ concatenated with $s$ be a candidate for $\kappa(v)$.
9.     for every unreduced sequence $v$ that has designated candidates in Steps 6-8
10.        do $\kappa(v) \leftarrow$ its first candidate in length-lexicographical order
11. until The last iteration has no new sequence defined for $\kappa$.
12. return $\kappa$
```

Algorithm 7 (*ChgResp5*) handles the change consequences incurred by mapping a previously legal and unreduced sequence $u$ to $\omega$. The sequence under change cannot be $\lambda$, as $\lambda$ must be mapped to 0 and its response cannot be changed. Algorithm 7 (*ChgResp5*) first calls Algorithm *CompMapForRespOrEquivChg* to get the mapping $\kappa$ from old unreduced sequences to new unreduced sequences after the change (Steps 1-2). Then it initializes $\mathcal{E}'$ with the empty sequence (Step 3) and explicitly defines new response for $u$ (Step 4). Next, every enumerated non-empty sequence in $\mathcal{E}$ is considered. If the corresponding transition remains in the modified automaton, it is translated into a row in $\mathcal{E}'$ (Steps 5-7). Finally, every old illegal and unreduced sequence is examined. If the same state is preserved but represented by a legal unreduced sequence in $\mathcal{E}'$, then the new unreduced sequence is explicitly extended (Steps 8-11).

```
Algorithm 7 (ChgResp5)($\mathcal{E}$, $u$)
Input: A complete and finite enumeration $\mathcal{E}$, an enumerated legal, unreduced sequence $u$ except $\lambda$
Output: A complete and finite enumeration $\mathcal{E}'$
(* Change a legal response to illegal for an unreduced *)
(* sequence *)
1. Initialize an empty hash map $\kappa$.
2. $\kappa \leftarrow CompMapForRespOrEquivChg(\mathcal{E}, u)$
3. Initialize $\mathcal{E}'$ to contain $\lambda$ only, with $\lambda$ mapped to 0 and unreduced.
4. Add sequence $u$ in $\mathcal{E}'$. Map $u$ to $\omega$ and make it unreduced.
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 $v$
6.     do if $ps \neq u$ 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 $\kappa(v)$.
```

```
 8. for every enumerated illegal and unreduced sequence
    $v$ in $\mathcal{E}$
 9.   do if $\kappa(v) \neq$ nil and $BlackBox(\mathcal{E}, \kappa(v)) \neq \omega$
10.     then for every stimulus $s$
11.       do Add the following sequence in $\mathcal{E}'$: prefix
             sequence $\kappa(v)$ followed by stimulus $s$ map-
             ped to $\omega$ and reduced to sequence $\kappa(v)$.
12. return $\mathcal{E}'$
```

## 3.4 Changing an equivalence for an unreduced legal sequence: Algorithm 9

Suppose we pick an enumerated legal, unreduced sequence $u$ and want to change its reduction. We assume the new reduced value of $u$ (denoted by $v$) does not contradict with any of the enumeration rules. If a sequence is unreduced, we treat its reduced value as the sequence itself. If a sequence is reduced to itself, it is essentially unreduced. Error checking for valid reductions is then separated from our major concern of the consequences of a possible equivalence change. In this case $v$ could be any unreduced prior sequence of $u$ in length-lexicographical order.

This case looks much like the above case of a response change. In both cases a formerly legal and unreduced sequence is not going to be extended in the new enumeration. The state it represents in the old automaton may or may not be preserved in the new automaton. In case it is preserved, it no longer represents the same unreduced sequence in the new enumeration. The same applies to any formerly unreduced sequence that happens to be an extension of this sequence $u$ under change. Unreduced sequences for $\mathcal{E}'$ need to be computed from the modified automaton globally.

Algorithm 7 can be slightly modified to solve this problem. Instead of adding a new trap state, we redirect the last transition incurred by $u$ to an existing state represented by $v$.

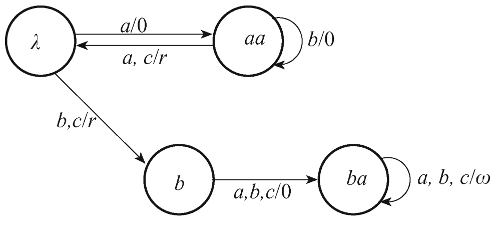
All implications of Algorithm 7 apply except for the third bulleted item. We have the following instead:

- Sequence $u$ is reduced to $v$ in $\mathcal{E}'$.

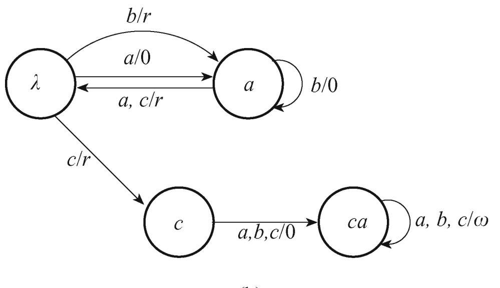
As an example Fig. 4 shows the state machines before and after changing the equivalence of a formerly unreduced, legal sequence $b$.

Algorithm 9 (*ChgEquiv2*) is very similar to Algorithm 7 (*ChgResp5*), except for Step 4 in which a new reduction rather than a new response is defined for the sequence $u$ under change.

**Algorithm** 9 (*ChgEquiv2*)($\mathcal{E}$, $u$, $v$)
**Input:** A complete and finite enumeration $\mathcal{E}$, an enumerated legal, unreduced sequence $u$ except $\lambda$, an enumerated unreduced sequence $v$ prior to $u$



(a)



(b)

**Fig. 4** Example automaton diagrams for changing an equivalence for an unreduced legal sequence. (a) before changing the equivalence of $b$, (b) after changing the equivalence of $b$

**Output:** A complete and finite enumeration $\mathcal{E}'$
(* Change equivalence for an unreduced legal sequence *)

```
 1. Initialize an empty hash map $\kappa$.
 2. $\kappa \leftarrow CompMapForRespOrEquivChg(\mathcal{E}, u)$
 3. Initialize $\mathcal{E}'$ to contain $\lambda$ only, with $\lambda$ mapped to 0 and
    unreduced.
 4. Add sequence $u$ in $\mathcal{E}'$. Map $u$ to its response in $\mathcal{E}$ and
    reduce it to $v$.
 5. for every enumerated sequence in $\mathcal{E}$ of the form: pre-
    fix sequence $p$ followed by stimulus $s$ mapped to re-
    sponse $r$ and reduced to sequence $w$
 6.   do if $ps \neq u$ 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 $s$ map-
             ped 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 $s$ map-
             ped to $\omega$ and reduced to sequence $\kappa(w)$.
12. return $\mathcal{E}'$
```

## 3.5 Time complexity analysis

Our change algorithms all contain as an input a complete and finite enumeration. They may contain other inputs such as a stimulus (for stimulus changes), a response (for response changes), or a stimulus sequence (for response/ equivalence changes), the size of which can be ignored compared with the size of the enumeration.

We will use a red-black tree to store an enumeration table. Every row of the table is stored with a node in the tree that has both a key and a value. The keys are for the enumerated sequences; the values are for the structures holding the associated responses, equivalences, and traces. The keys (sequences) are sorted length-lexicographically.

Let $n$ be the size of a complete and finite enumeration, and the input size of any of the change algorithms. Then the number of enumerated sequences (the number of rows, or the number of nodes in the red-black tree) is bounded by $n$. It follows that stimulus addition (Algorithm *1* (*AddStim*)) takes time $O(n \log(n))$, as each tree operation executes in $\log(n)$ time.

For stimulus deletion (Algorithm *2* (*DelStim*)), we first examine how long it takes to compute the partial function $\kappa$ using Algorithm *CompMapForStimDel*. Finding all unreduced sequences takes time $O(n \log(n))$. The number of all unreduced sequences is bounded by $n$. The first **for** loop (Lines 2-4) executes in time $O(n \log(n))$, and the following **repeat-until** loop (Lines 5-11) is executed at most $n$ times, as each iteration identifies the $\kappa$ mapping for at least one unreduced sequence in the original enumeration. In each iteration, the red-black tree is traversed in time $O(n \log(n))$ (Line 6). For each node visit $O(\log(n))$ time is needed for looking up values in the $\kappa$ mapping as well as for computing the mapped response for $\kappa(p)$ (Lines 7-8). The **for** loop in Lines 9-10 takes time $O(n \log(n))$. Therefore, the time complexity for Algorithm *CompMapForStimDel* is

$$O(n \ \log(n)) + n^*(O(n \ \log(n)) + O(n \ \log(n))) \\ = O(n^2 \ \log(n)).$$

Now consider Algorithm *2* (*DelStim*). Line 1 takes constant time $O(1)$. Line 2 takes time $O(n^2 \log(n))$. Line 3 takes time $O(\log(n))$. The **for** loop in Lines 4-6 is executed at most $n$ times each taking time $O(\log(n))$. The **for** loop in Lines 7-10 also executes in time $O(n \log(n))$. Hence the total running time is $O(n^2 \log(n))$.

Similarly we can prove that among all response and equivalence change algorithms, only two of them, i.e., Algorithms *7* (*ChgResp5*) and *9* (*ChgEquiv2*), take time $O(n^2 \log(n))$; the others all have time complexity $O(n \log(n))$.

All the twelve change algorithms are polynomial-time algorithms.

## 4 Example: safe controller with security alarm

Suppose our safe controller product introduced in Section 2 is now enhanced with a new feature. On a wrong combination entry, a security alarm will be triggered, which cannot be cleared until the correct three digits are entered. This will replace tagged requirement 7 in Table 1, and be tagged as 7′.

Two new outputs across the system boundary are identified: *alarm on* and *alarm off*. With our change algorithms, we do not need to start over re-enumerating, but instead can build on the previous enumeration (Table 4) following the atomic steps listed in Table 6.

Among the twelve atomic steps, only the first two steps are needed to clear things up in the old safe enumeration (Table 4), and preserve the work that has already been completed. The rest of the steps (Steps 3-12) are needed anyway to specify the new behavior paired with the newly introduced requirement 7′; the work cannot be saved even if one re-enumerates the new system from the beginning.

A completed specification obtained for the enhanced safe controller is shown in Table 7.

If the requirements changes go in the other direction and we start with the enumeration for the enhanced product, it takes only two atomic requirements changes to derive a specification of the original product, as shown in Table 8. All the change consequences will be taken care of by the available change algorithms and the tool support.

**Table 6** Atomic steps needed for adding the alarm feature

| Step | Algorithm | Atomic change |
|---|---|---|
| 1 | Algorithm 3 | change the response of *LB* from 0 to *alarm on* |
| 2 | Algorithm 12 | change the equivalence of *LBC* from *L* to *LBC* |
| 3 | Algorithm 5 | change the response of *LBCB* from $\omega$ to 0 |
| 4 | Algorithm 9 | change the equivalence of *LBCB* from *LBCB* to *LB* |
| 5 | Algorithm 5 | change the response of *LBCC* from $\omega$ to 0 |
| 6 | Algorithm 9 | change the equivalence of *LBCC* from *LBCC* to *LBC* |
| 7 | Algorithm 5 | change the response of *LBCG* from $\omega$ to *alarm off*, *unlock* |
| 8 | Algorithm 9 | change the equivalence of *LBCG* from *LBCG* to *U* |
| 9 | Algorithm 5 | change the response of *LBCL* from $\omega$ to 0 |
| 10 | Algorithm 9 | change the equivalence of *LBCL* from *LBCL* to *L* |
| 11 | Algorithm 5 | change the response of *LBCU* from $\omega$ to 0 |
| 12 | Algorithm 9 | change the equivalence of *LBCU* from *LBCU* to *U* |

**Table 7** Enhanced safe controller sequence enumeration

| Sequence | Response | Equivalence | Trace |
|---|---|---|---|
| $\lambda$ | 0 | | Method |
| $B$ | $\omega$ | | D1 |
| $C$ | $\omega$ | | D1 |
| $D$ | $\omega$ | | D1 |
| $G$ | $\omega$ | | D1 |
| $L$ | 0 | | 5 |
| $U$ | 0 | | 5 |
| $LB$ | *alarm on* | | 1,2,7′ |
| $LC$ | 0 | $L$ | 2 |
| $LD$ | $\omega$ | | 8 |
| $LG$ | *unlock* | $U$ | 1,3 |
| $LL$ | 0 | $L$ | 5,D2 |
| $LU$ | 0 | $U$ | 5,D2 |
| $UB$ | 0 | $U$ | 6 |
| $UC$ | 0 | $U$ | 6 |
| $UD$ | *lock* | $L$ | 4 |
| $UG$ | 0 | $U$ | 6 |
| $UL$ | 0 | $L$ | 5,D2 |
| $UU$ | 0 | $U$ | 5,D2 |
| $LBB$ | 0 | $LB$ | 2 |
| $LBC$ | 0 | | 2 |
| $LBD$ | $\omega$ | | 8 |
| $LBG$ | 0 | $LB$ | 2 |
| $LBL$ | 0 | $L$ | 5,D2 |
| $LBU$ | 0 | $U$ | 5,D2 |
| $LBCB$ | 0 | $LB$ | 2 |
| $LBCC$ | 0 | $LBC$ | 2 |
| $LBCD$ | $\omega$ | | 8 |
| $LBCG$ | *alarm off, unlock* | $U$ | 1,3,7′ |
| $LBCL$ | 0 | $L$ | 5,D2 |
| $LBCU$ | 0 | $U$ | 5,D2 |

**Table 8** Atomic steps needed for removing the alarm feature

| Step | Algorithm | Atomic change |
|---|---|---|
| 1 | Algorithm 3 | change the response of $LB$ from *alarm on* to 0 |
| 2 | Algorithm 9 | change the equivalence of $LBC$ from $LBC$ to $L$ |

As an example, we show below how Algorithm 9 (*ChgEquiv2*) is applied at the last step to produce Table 4. The enumeration after executing Step 1 of Table 8 is almost the same as the enhanced safe enumeration except that $LB$ is mapped to 0. Then among the six unreduced sequences $\lambda$, $B$, $L$, $U$, $LB$, and $LBC$, only $LBC$ contains the sequence under change (i.e., $LBC$) as a prefix. The $\kappa$ mapping from old unreduced sequences to new unreduced sequences is computed as follows:

$$\begin{aligned}\kappa(\lambda) &= \lambda\\ \kappa(B) &= B\\ \kappa(L) &= L\\ \kappa(U) &= U\\ \kappa(LB) &= LB\\ \kappa(LBC) &= \mathbf{nil}\end{aligned}$$

Sequence $LBC$ is not defined for $\kappa$ as the only two sequences reduced to $LBC$ (sequences $LBC$ and $LBCC$) do not satisfy the condition required by Line 7 of Algorithm *CompMapForRespOrEquivChg*, hence no candidate values for $\kappa(LBC)$ are found. Most of the entries in the enhanced safe enumeration can be translated to corresponding entries in the safe enumeration with the $\kappa$ function. Sequences with $LBC$ as a proper prefix (the last six sequences) are removed since $\kappa(LBC) = \mathbf{nil}$ and it breaks the condition required by Line 6 of Algorithm 9 (*ChgEquiv2*).

## 5 Conclusion and related work

Requirements changes can occur both during and after a phase of development. Algorithms for managing requirements changes meet a very great need in field application of the sequence-based specification method. In this paper we propose to capture requirements changes as a series of atomic specification changes, categorize all possible atomic requirements changes, and propose algorithms for managing them. All the change algorithms have been implemented in the prototype tool supporting sequence-based specification. They are also supported by rigorous mathematical formulation and proof of correctness [7].

Changing requirements has long been recognized as a major source of risk and the cause for many difficult and costly errors during the software development life cycle [10–13]. Various research has been conducted focusing on process improvement, process modeling and measurement, and change effort (cost) estimation [14,15]. For instance, in [16] a model for requirements change management was proposed based on the needs of stakeholders and the relationships between objects (e.g., needs, goals, actions), however, it remains unclear how requirements changes (and changes of needs) translate into behavioral changes explicit and detailed enough for implementation.

A lot of research on change impact analysis focuses on the code level [17–20]. Lock and Kotonya [21] proposed an approach supporting change impact analysis at the requirements level, using probability to assist in the combination and presentation of predicted impact propagation paths. They, too, focused on functional requirements of a system, but limited traceability analysis to within requirements level artifacts. The impact of a requirements change is determined using other related requirements without mapping requirements changes to changes in specifications or the code.

Our work differs from the previous work in that it strives to provide a general solution to requirements change management independent of the application domain, and a rigorous approach that precisely prescribes the impact of requirements changes on the specification (and the code), with software development and maintenance treated as a mathematical process. Since every problem that can be programmed for computers can be modeled as a finite state machine, and sequence-based specification systematically discovers the state machine from informal requirements, the work presented in this paper provides a rigorous change management theory and automation concepts needed to avoid the high risk of introducing errors (as compared with changing specification and code manually).

Our theory of managing requirements changes in sequence-based specifications differs from the conventional state change theory in that it is designed for active state machine development and revision. For example, Korel [22] presented an approach of understanding model-based modifications that uses the original model and the modified model to compute the effect of the modifications through affecting and affected transitions. It assumes the availability of the modified model and bases the analysis on data and control dependence among inputs and outputs. The change might be the result of maintenance, error correction, or change in functionality driven by change in requirements. However, these algorithms would not support an original enumeration process, or complete the revision of changes in an enumeration.

Seawright and Brewer [23] presented "production-based specification" and used a similar base of language-automata theory to convert grammar productions into hardware design. They, too, focus on external behavior relative to a well-defined system boundary and its interfaces, and generate Mealy-Moore state machine descriptions as an intermediate step toward circuit design. Although our algorithms could apply to production-based specification at their intermediate state machine, it is not clear how such changes would reflect in their original productions. Their production language is quite powerful, subject to debugging, and describes a larger class of Mealy machines than those represented by enumerations.

Future research along this line includes abstraction change management, in which the theoretical basis for introducing and removing abstractions as well as shifting up and down in levels of abstractions will be explored.

## References

1. Prowell S J, Poore J H. Sequence-based software specification of deterministic systems. Software – Practice and Experience, 1998, 28(3): 329–344
2. Prowell S J, Poore J H. Foundations of sequence-based software specification. IEEE Transactions on Software Engineering, 2003, 29(5): 417–429
3. Mills H D. The new math of computer programming. Communications of the ACM, 1975, 18(1): 43–48
4. Mills H D. Stepwise refinement and verification in box-structured systems. IEEE Computer, 1988, 21(6): 23–36
5. Prowell S J, Trammell C J, Linger R C, et al. Cleanroom Software Engineering: Technology and Process. Addison-Wesley-Longman, 1999
6. Bauer T, Beletski T, Boehr F, et al. From requirements to automated testing of quasar aussenspiegeleinstellung. Technical Report 007.07E, Fraunhofer Institute for Experimental Engineering, 2007
7. Lin L. Management of Requirements Changes in Sequence-Based Software Specifications. Dissertation for the Doctoral Degree. University of Tennessee, 2006. http://sqrl.cs.utk.edu/btw/files/lin.pdf
8. Hopcroft J E, Ullman J D. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979
9. Lin L, Prowell S J, Poore J H. Management of requirements changes in sequence-based specifications. Technical Report ut-cs-07-588, University of Tennessee, 2007
10. Buren J, Cook D. Experiences in the adoption of requirements engineering technologies. Crosstalk: The Journal of Defense Software Engineering, 1998, December: 3–10
11. Davis A. Software Requirements: Analysis and Specification. Prentice-Hall, 1989
12. Harker S, Eason K, Dobson J. The change and evolution of requirements as a challenge to the practice of software engineering. In: Proceedings of the IEEE International Symposium on Requirements Engineering, 1993, 266–272
13. Jones C. Strategies for managing requirements creep. IEEE Computer, 1996, 29(6): 92–94
14. Lavazza L, Valetto G. Enhancing requirements and change management through process modeling and measurement. In: Proceedings of the Fourth International Conference on Requirements Engineering, 2000, 106–115
15. Nurmuliani N, Zowghi D, Williams S. Requirements volatility and its impact on change effort: Evidence-based research in software development projects. In: Proceedings of the Eleventh Australian Workshop on Requirements Engineering, 2006
16. Kobayashi A, Maekawa M. Need-based requirements change management. In: Proceedings of the Eighth Annual IEEE International Conference and Workshop on Engineering of Computer-Based Systems, 2001, 171–178
17. Ajila S. Software maintenance: An approach to impact analysis of objects change. Software – Practice and Experience, 1995, 25(10): 1155–1181
18. Lee M, Offutt A, Alexander R. Algorithmic analysis of the impacts of changes to object-oriented software. In: Proceedings of the Thirty-Fourth International Conference on Technology of Object-Oriented Languages and Systems, 2000, 61–70
19. Moriconi M, Winkley T. Approximate reasoning about the effects of program changes. IEEE Transactions on Software Engineering, 1990, 16(9): 980–992
20. Ren X, Barbara G, Maximilian S, et al. Chianti: A change impact analysis tool for Java programs. In: Proceedings of the Twenty-Seventh International Conference on Software Engineering, 2005, 61–70
21. Lock S, Kotonya G. An integrated, probabilistic framework for requirements change impact analysis. Australian Journal of Information Systems, 1999, 6(2): 38–63
22. Korel B, Tahat L H. Understanding modifications in state-based models. In: Proceedings of the Twelfth IEEE International Workshop on Program Comprehension. IEEE Computer Society Press, 2004.
23. Seawright A, Brewer F. Clairvoyant: A synthesis system for production-based specification. IEEE Transactions on VLSI Systems, 1994, 2(2): 172–185