Quasi-Linearizability is Undecidable

Chao Wang[1,2(✉)], Yi Lv[1], Gaoang Liu[1,2], and Peng Wu[1]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{wangch,lvyi,gaoang,wp}@ios.ac.cn

[2] University of Chinese Academy of Sciences, Beijing, China

Abstract. *Quasi-linearizability* is a quantitative relaxation of linearizability. It preserves the intuition of the standard notion of linearizability and permits more flexibility. The decidability of quasi-linearizability has been remaining open in general for a bounded number of processes. In this paper we show that the problem of whether a library is *quasi-linearizable* with respect to a regular sequential specification is undecidable for a bounded number of processes. This is proved by reduction from the k-Z decision problem of a k-counter machine, a known undecidable problem. The key idea of the proof is to establish a correspondence between the quasi-sequential specification of quasi-linearizability and the set of all unadmitted runs of the k-counter machines.

1 Introduction

A concurrent library provides a collection of methods for accessing a concurrent object. These methods can be invoked by multiple client processes concurrently. Linearizability [11] is a *de facto* correctness condition for concurrent libraries. A concurrent library is linearizable with respect to its sequential specification, if during any of its executions, each method appears to take effect instantaneously at some point between the invocation and the response of the method.

The standard notion of linearizability imposes a strong synchronization requirement that, in many cases, leads to performance and scalability bottlenecks and hence prevents effective utilization of increasingly parallel hardware. A remedy to this problem is to relax this consistency condition. Although there already exist other consistency conditions for concurrent libraries, such as sequential consistency [12] and quiescent consistency [4], these consistency conditions are less intuitive and allow unexpected behaviors. New relaxed consistency conditions have been proposed recently, including *quasi-linearizability* [2] and a quantitative relaxation framework [9]. These relaxed consistency conditions are essentially based on linearizability, therefore preserving the intuition of the standard linearizability while permitting more flexibility.

This work is partially supported by the National Natural Science Foundation of China under Grants No.60721061, No.60833001, No.61272135, No.61700073, No.61100069, No.61472405, and No.61161130530.

X. Feng and S. Park (Eds.): APLAS 2015, LNCS 9458, pp. 369–386, 2015.
DOI: 10.1007/978-3-319-26529-2_20

Quasi-linearizability is the first quantitative relaxation of linearizability. It extends the standard linearizability by relaxing the sequential specification to a *Q-quasi-sequential specification*. Such quantitative relaxation is guided by a *quasi-linearization factor Q*. Each element in the Q-quasi-sequential specification is a bounded distance away from a legal one. Therefore, the verification of quasi-linearizability needs to deal with not only the subtle difficulty of linearizability, but also that of the relaxed sequential specification.

It is well known that the problem of whether a library is linearizable with respect to its regular sequential specification is decidable for a bounded number of processes [3], but undecidable for an unbounded number of processes [5]. Since the standard linearizability is a special case of quasi-linearizability, it can be easily seen that the problem of whether a library is quasi-linearizable with respect to its regular sequential specification is also undecidable for an unbounded number of processes. For the case with a bounded number of processes, [1,13] have presented model-checking algorithm and runtime tool respectively to check quasi-linearizability with specific quasi-linearization factors. However, the problem of whether a library is quasi-linearizable with respect to a regular sequential specification still remains open in general for a bounded number of processes.

The main result of this paper is to show that the problem of whether a library is quasi-linearizable with respect to a regular sequential specification (quasi-linearizability problem) is undecidable for a bounded number of processes. Our proof can be divided into two parts.

In the first part we reduce the k-Z decision problem of a k-counter machine [3] to the problem of whether a specific concurrent library is linearizable with respect to its sequential specification (linearizability problem) for just one process. The k-Z problem is to decide whether a k-counter machine has an admitted run. This problem is known to be undecidable for $k \geq 3$ [3]. As inspired by the proof of Lemma 5 in [3], the k-Z decision problem can be reduced to a language inclusion problem between two language R and S. R is a prefix-closed regular language constructed from the k-counter machine. S is a non-regular language and it contains all the unadmitted runs of k-counter machine, together with their prefixes. A specific library $\mathcal{L}_R$ can then be constructed to simulate sequentially each sequence of R. Thus, the language inclusion problem can be reduced to the linearizability problem of $\mathcal{L}_R$ for just one process with respect to a non-regular sequential specification constructed from S.

In the second part we prove that the above linearizability problem can then be reduced to a quasi-linearizability problem of the same library for just one process. The quasi-linearizability problem uses a regular sequential specification and a proper quasi-linearization factor. Since the quasi-linearizability problem of the specific library $\mathcal{L}_R$ for one process is equivalent to that for a bounded number of processes, the k-Z decision problem is further reduced to quasi-linearizability problem with respect to a regular sequential specification for a bounded number of processes.

Related Work. There are already several publications on the decidability of linearizability and other consistency conditions [3,5,6,8], two of which are related to our work.

Bouajjani *et al.* [5] proved that the problem of whether a library is linearizable with respect to its regular sequential specification is undecidable for an unbounded number of processes. This was proved through a reduction from an undecidable problem of a counter machine.

The closer work to ours is [3] by Alur *et al.*, which proves that the linearizability of a regular history set with respect to a regular sequential specification is decidable for a bounded number of processes, but its sequential consistency is not. The proofs in [3] rely on the notion of the language closure over a dependency relation. Given a binary dependency relation D over an alphabet Σ, the closure of a language $L \subseteq \Sigma^*$ is the set of all sequences that are obtained from a sequence $l \in L$ by shuffling any adjacent symbols a and b in l such that $(a, b) \notin D$. It may sound possible to encode a dependency relation directly by a quasi-linearization factor Q, which allows shuffling of adjacent independent symbols within certain bound. In this way, the undecidability of quasi-linearizability could be regarded as a corollary of Lemma 5 in [3].

However, unfortunately, there exists a dependency relation that can not be characterized by any quasi-linearization factor, e.g., a dependency relation for sequential consistency. This is because that sequential consistency permits shuffling only symbols of different processes and hence does not allow any shuffling when a history contains only one process. In contrast, relaxations of quasi-linearizability are irrelevant to the number of processes a history may contain. Thus, if a quasi-linearization factor Q permits shuffling symbols of more than one process, it should also permit shuffling symbols of a single process. Based on this observation, the undecidable result of Lemma 5 in [3] can not be directly applied to establish the undecidability of quasi-linearizability.

Adhikari *et al.* [1] proposed a model-checking algorithm for verification of the quasi-linearizability that uses a specific quasi-linearization factor. Zhang *et al.* [13] developed a runtime tool to verify quasi-linearizability with respect to the relaxed sequential specifications that are based on the *strict out-of-order semantics* defined in [2,9]. Both works consider only decidable subclasses of the quasi-linearizability problem.

2 Concurrent Systems

In this section, we present the notations of libraries, the most general clients and concurrent systems. We then introduce their operational semantics.

2.1 Notations

A finite sequence on an alphabet Σ is denoted as $l = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_k$, where $\cdot$ is the concatenation symbol and $\alpha_i \in \Sigma$ for each $1 \leq i \leq k$. For an alphabet Σ', let $l \uparrow_{\Sigma'}$ denote the projection of l to Σ'. Given a set S of sequences, we use $\mathit{Prefix}(S) = \{a_1 \cdot \ldots \cdot a_m | \exists u$ and $a_{m+1}, \ldots, a_{m+u}$, such that $a_1 \cdot \ldots \cdot a_{m+u} \in S\}$ to denote the set of prefixes of sequences in S. Given a function f, let $f[x : y]$ be the function that shares the same value as f everywhere, except for x, where it has the value y. Given a function $f : \mathcal{A} \rightarrow \mathcal{B}$, we use $domain(f)$ to denote the domain of f, which is $\mathcal{A}$. We use _ for an item, of which the value is irrelevant.

A *labelled transition system* (LTS) is a tuple $\mathcal{A} = (Q, \Sigma, \rightarrow, q_{init})$, where Q is a set of states, Σ is a set of transition labels, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation and q_{init} is the initial state. A state of the LTS $\mathcal{A}$ can be refer to as a *configuration* in the rest of the paper. A path of $\mathcal{A}$ is a finite transition sequence $q_{init} \xrightarrow{\beta_1} q_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_k} q_k$ for $k \geq 0$ from the initial state q_{init}. A trace of $\mathcal{A}$ is a finite sequence $t = \beta_1 \cdot \beta_2 \cdot \ldots \cdot \beta_k$, where $k \geq 0$ if there exists a path $q_{init} \xrightarrow{\beta_1} q_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_k} q_k$ of $\mathcal{A}$.

2.2 Libraries and the Most General Clients

A library implementing a concurrent object that provides a set of methods for external users to access the data structure. It may contain private memory locations for its own use. A client program is a program that interacts with libraries. For simplicity, we assume that each method has just one parameter and one return value if it returns. Furthermore, all the parameters and the return values are passed via a specific register r_f.

For a library, let $\mathcal{X}$ be a finite set of its memory locations, $\mathcal{M}$ be a finite set of its method names, $\mathcal{D}$ be its finite data domain, $\mathcal{R}$ be a finite set of its register names and $\mathcal{RE}$ be a finite set of its register expressions over $\mathcal{R}$. Then, a set $PCom$ of primitive commands of the library includes:

- Register assign commands in the form of $r = re$;
- Register reset commands in the form of *havoc*;
- Read commands in the form of *read* (x, r) ;
- Write commands in the form of $write(r, x)$;
- *Cas* commands in the form of $r_1 = cas(x, r_2, r_3)$;
- Assume commands in the form of $assume(r)$;
- Call commands in the form of $call(m)$;

where $r, r_1, r_2, r_3 \in \mathcal{R}, re \in \mathcal{RE}, x \in \mathcal{X}$. Herein, the notations of registers and register expressions are similar to those used in [7]. A *cas* command compresses a read and a write commands into a single one, which is meant to be executed atomically. It is often implemented with the compare-and-swap or load-linked/store-conditional primitive at the level of multiprocessors. This type of commands is widely used in concurrent libraries. A *havoc* command [7] assigns arbitrary values to all registers in $\mathcal{R}$.

A control-flow graph is a tuple $CFG = (N, L, T, q_i, q_f)$, where N is a finite set of program positions, L is a set of primitive commands, $T \subseteq N \times L \times N$ is a control-flow transition relation, q_i is the initial position and q_f is the final position.

A library $\mathcal{L}$ can then be defined as a tuple $\mathcal{L} = (Q_{\mathcal{L}}, \rightarrow_{\mathcal{L}}, InitV_{\mathcal{L}})$, such that $Q_{\mathcal{L}} = \bigcup_{m \in \mathcal{M}} Q_m$ is a finite set of program positions, where Q_m is the program positions of a method m of this library; $\rightarrow_{\mathcal{L}} = \bigcup_{m \in \mathcal{M}} \rightarrow_m$ is a control-flow transition relation, where for each $m \in \mathcal{M}$, $(Q_m, PCom, \rightarrow_m, i_m, f_m)$ is a control-flow graph with a unique initial position i_m and a unique final position f_m; $InitV_{\mathcal{L}} : \mathcal{X} \rightarrow \mathcal{D}$ is an initial valuation for its memory locations.

The most general client of a library is a special client program that is used to exhibit all possible behavior of the library. Formally, the most general client $\mathcal{MGC}$ of library $\mathcal{L}$ is defined as a tuple $(\{q_c, q'_c\}, \rightarrow_c)$, where q_c and q'_c are two program positions, $\rightarrow_c = \{(q_c, havoc, q'_c)\} \cup \{(q'_c, call(m), q_c) | m \in \mathcal{M}\}$ is a control-flow transition relation and $(\{q_c, q'_c\}, PCom, \rightarrow_c, q_c, q_c)$ is a control-flow graph. Intuitively, the most general client repeatedly calls an arbitrary method with an arbitrary argument for arbitrarily many times.

2.3 Operational Semantics of Concurrent Systems

In this paper we consider a concurrent system consists of a bounded number of processes, each of which runs the most general client program of a library on a separate processor. Then, the operational semantics of a library can be defined in the context of the concurrent system.

For a library $\mathcal{L}$=$(Q_{\mathcal{L}}, \rightarrow_{\mathcal{L}}, InitV_{\mathcal{L}})$ and a positive integer n, its operational semantics is defined as an LTS $[\![\mathcal{L}, n]\!]_{cs} = (Conf_{cs}, \Sigma_{cs}, \rightarrow_{cs}, InitConf_{cs})$, where *'cs'* represents concurrent system, and $Conf_{cs}, \Sigma_{cs}, \rightarrow_{cs}, InitConf_{cs}$ are defined as follows.

Each configuration of $Conf_{cs}$ is a tuple (p, d, r), where

– $p : \{1, \ldots, n\} \rightarrow \{q_c, q'_c\} \cup Q_{\mathcal{L}}$ represents control states of each process;
– $d : \mathcal{X} \rightarrow \mathcal{D}$ represents values at each memory location;
– $r : \{1, \ldots, n\} \rightarrow (\mathcal{R} \rightarrow \mathcal{D})$ represents values of the registers of each process.

Σ_{cs} consists of the following subsets of actions as transition labels.

– Internal actions: $\{\tau(i) | 1 \leq i \leq n\}$;
– Read actions: $\{read(i, x, a) | 1 \leq i \leq n, x \in \mathcal{X}, a \in \mathcal{D}\}$;
– Write actions: $\{write(i, x, a) | 1 \leq i \leq n, x \in \mathcal{X}, a \in \mathcal{D}\}$;
– *Cas* actions: $\{cas(i, x, a, b) |\ 1 \leq i \leq n, x \in \mathcal{X}, a, b \in \mathcal{D}\}$;
– Call actions: $\{call(i, m, a) | 1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}\}$;
– Return actions: $\{return(i, m, a) | 1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}\}$;

The initial configuration $InitConf_{cs} \in Conf_{cs}$ is a tuple $(p_{init}, InitV_{\mathcal{L}}, r_{init})$, where $p_{init}(i) = q_c$ and $r_{init}(i)(r) = regV_{init}$ (a specific initial value of register) for $1 \leq i \leq n$, $r \in \mathcal{R}$;

The transition relation $\rightarrow_{cs}$ is the least relation satisfying the transition rules shown in Fig. 1.

– *Register-Assign* rule: A function $f_{re} : (\mathcal{R} \rightarrow \mathcal{D}) \times \mathcal{RE} \rightarrow \mathcal{D}$ is used to evaluate register expression re under register valuation rv of current process, and its value is assigned to register r_1.
– *Library-Havoc* and $\mathcal{MGC}$*-Havoc* rules: *havoc* commands are executed for libraries and the most general clients respectively.
– *Assume* rule: If the value of register r_1 is *true*, current process can execute *assume* command. Otherwise, it must wait.

– *Read* and *Write* rules: A read action to memory location x will take the value of x in memory, and a write action to memory location x will change the value of x in memory directly.
– *Cas-Success* and *Cas-Fail* rules: A successful *cas* command will change the value of memory location x immediately. The result of whether this *cas* command succeeds is stored in register r_1.
– *Call* and *Return* rules: To deal with *call* command, current process starts to execute the initial position of method m. When the process comes to the final position of method m it can launch a return action and start to execute the most general client.

$$\frac{p(i) = q_1, q_1 \xrightarrow{r_1 = re}_{\mathcal{L}} q_2, r(i) = rv, f_{re}(rv, re) = a}{(p, d, r) \xrightarrow{\tau(i)}_{cs} (p[i : q_2], d, r[i : rv[r_1 : a]])} \textit{Register-Assign}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{havoc}_{\mathcal{L}} q_2, rv \in \mathcal{R} \rightarrow \mathcal{D}}{(p, d, r) \xrightarrow{\tau(i)}_{cs} (p[i : q_2], d, r[i : rv])} \textit{Library-Havoc}$$

$$\frac{p(i) = q_c, rv \in \mathcal{R} \rightarrow \mathcal{D}}{(p, d, r) \xrightarrow{\tau(i)}_{cs} (p[i : q_c'], d, r[i : rv])} \mathcal{MGC}\textit{-Havoc}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{assume(r_1)}_{\mathcal{L}} q_2, r(i)(r_1) = true}{(p, d, r) \xrightarrow{\tau(i)}_{cs} (p[i : q_2], d, r)} \textit{Assume}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{read(x, r_1)}_{\mathcal{L}} q_2, r(i) = rv, d(x) = a}{(p, d, r) \xrightarrow{read(i, x, a)}_{cs} (p[i : q_2], d, r[i : rv[r_1 : a]])} \textit{Read}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{write(r_1, x)}_{\mathcal{L}} q_2, r(i)(r_1) = a}{(p, d, r) \xrightarrow{write(i, x, a)}_{cs} (p[i : q_2], d[x : a], r)} \textit{Write}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{r_1 = cas(x, r_2, r_3)}_{\mathcal{L}} q_2, r(i) = rv, rv(r_2) = d(x) = a, rv(r_3) = b}{(p, d, r) \xrightarrow{cas(i, x, a, b)}_{cs} (p[i : q_2], d[x : b], r[i : rv[r_1 : true]])} \textit{Cas-Success}$$

$$\frac{p(i) = q_1, q_1 \xrightarrow{r_1 = cas(x, r_2, r_3)}_{\mathcal{L}} q_2, r(i) = rv, rv(r_2) = a, rv(r_3) = b, rv(r_2) \neq d(x)}{(p, d, r) \xrightarrow{cas(i, x, a, b)}_{cs} (p[i : q_2], d, r[i : rv[r_1 : false]])} \textit{Cas-Fail}$$

$$\frac{p(i) = q_c', r(i)(r_f) = a}{(p, d, r) \xrightarrow{call(i, m, a)}_{cs} (p[i : i_m], d, r)} \textit{Call}$$

$$\frac{p(i) = f_m, r(i)(r_f) = a}{(p, d, r) \xrightarrow{return(i, m, a)}_{cs} (p[i : q_c], d, r)} \textit{Return}$$

Fig. 1. Transition rules of $\rightarrow_{cs}$

3 Linearizability and Quasi-Linearizability

In this section, we introduce the definitions of linearizability and quasi-linearizability.

3.1 Linearizability

Linearizability is a standard correctness condition for concurrent libraries. According to [11], linearizability is a local property in the sense that a concurrent program that contains multiple concurrent libraries and client processes does not violate linearizability if each individual library does not violate linearizability. Therefore, it is safe for us to introduce the definition of linearizability using the operational semantics $[\![\mathcal{L}, n]\!]_{cs}$, which consider the behavior of only one library.

The behavior of a library is typically represented by histories of interactions between library and the clients calling it (through call and return actions). Let Σ_{cal} and Σ_{ret} represent the sets of all call and return actions, respectively. Given an *LTS* $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma_{\mathcal{A}}, \rightarrow_{\mathcal{A}}, q_{\mathcal{A}})$, a finite sequence $h \in (\Sigma_{cal} \cup \Sigma_{ret})^*$ is a history of $\mathcal{A}$ if there exists a trace t of $\mathcal{A}$ such that $t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})} = h$. Let *history*$(\mathcal{A})$ denote all the histories of $\mathcal{A}$.

In a history, a return action *return*(i_1, m_1, a_1) matches a call action *call*(i_2, m_2, a_2), if $i_1 = i_2 \wedge m_1 = m_2$. A history is sequential if it starts with a call action and each call (respectively, return) action is immediately followed by a matching return (respectively, a call) action unless it is the last action. A process subhistory $h|_i$ is a history consisting of all and only the actions of process i. A history h is *well-formed*, if each process subhistory $h|_i$ of h is sequential. All histories considered in this paper are assumed to be well-formed. Two histories h_1 and h_2 are equivalent, if for each process i, $h_1|_i = h_2|_i$. Given a history h, *complete*(h) is the maximal subsequence of h consisting of all matching call and return actions. An operation e in a history is a pair consisting of a call action, *inv*(e), and the next matching return action, *res*(e).

A sequential specification of a library is a prefix closed set of sequential histories. A history h is *linearizable* with respect to a sequential specification S, if h can be extended (by appending zero or more return actions) to a history h', and there exists a sequential history $s \in S$, such that

- *complete*(h') is equivalent to s.
- For each operations e_1, e_2 of h, if *res*(e_1) precedes *inv*(e_2) in h, then this also holds in s.

Definition 1 (Linearizability [11]). *A library $\mathcal{L}$ is linearizable with respect to a sequential specification S for n processes, if each history of $[\![\mathcal{L}, n]\!]_{cs}$ is linearizable with respect to S.*

It is natural to assume that for a sequential history *call*$(i_1, m_1, a_1) \cdot$ *return*$(i_1, m_1, b_1) \cdot \ldots \cdot$ *call*$(i_u, m_u, a_u) \cdot$ *return*(i_u, m_u, b_u) in a sequential specification, each process id i_j $(1 \leq j \leq u)$ is actually irrelevant. Thus we can substitute each pair of a call action $\mathit{call}(i, m, a)$ and its matching return action $\mathit{return}(i, m, b)$ with $m(a, b)$. For a library $\mathcal{L}$, let $\Sigma_{spec} = \{m(a, b) | \ m \in \mathcal{M}, a, b \in \mathcal{D}\}$, a sequential specification for $\mathcal{L}$ can also be given as a prefix closed subset of Σ^*_{spec}, as shown in [2].

Then, the notion of linearizability can be accordingly redefined over Σ^*_{spec}. A history h is linearizable with respect to a sequential specification $S \subseteq \Sigma^*_{spec}$, if there exists $m_1(a_1, b_1) \cdot \ldots \cdot m_u(a_u, b_u) \in S$, h can be extended (by appending zero or more return actions) to a history h', and there is a sequential history $s = \mathit{call}(i_1, m_1, a_1) \cdot \mathit{return}(i_1, m_1, b_1) \cdot \ldots \cdot \mathit{call}(i_u, m_u, a_u) \cdot \mathit{return}(i_u, m_u, b_u)$, such that

- *complete*(h') is equivalent to s.
- For each operation e_1, e_2 of h, if *res*(e_1) precedes *inv*(e_2) in h, then this also holds in s.

A library $\mathcal{L}$ is *linearizable* with respect to a sequential specification $S \subseteq \Sigma^*_{spec}$ for n processes, if each history of $[\![\mathcal{L}, n]\!]_{cs}$ is linearizable with respect to S. To comply with the definitions in [2], all the sequential specifications in the rest of this paper are prefix closed subsets of Σ^*_{spec}. Given a library $\mathcal{L}$, a sequential specification S and a positive integer n, the decision problem of linearizability is to determine whether $\mathcal{L}$ is linearizable with respect to S for n processes.

3.2 Quasi-Linearizability

Quasi-Linearizability [2] is a quantitative relaxation of linearizability. Quasi-linearizablity is also a local property [2]. Hence, as in the previous subsection, it is safe for us to introduce the definition of quasi-linearizability using the operational semantics $[\![\mathcal{L}, n]\!]_{cs}$.

For each element α in a sequence l, we use $l[\alpha]$ to denote its index. Given two sequences $l_1, l_2 \in \Sigma^*_{spec}$, where l_1 is a permutation of l_2, we use *distance*(l_1, l_2) $=$ *max*$\{l_1[\alpha] - l_2[\alpha] | \ \alpha$ is in $l_1\}$ to represent the distance between l_1 and l_2. A *quasi-linearization factor* Q is a function defined as $Q : D \rightarrow \mathcal{N}$, where D is a subset of the power set of Σ_{spec}. A quasi-linearization factor is used to guide the relaxation of quasi-linearizability. Given a sequential specification $S \subseteq \Sigma^*_{spec}$ and a quasi-linearization factor Q, the *Q-quasi-sequential specification* *Q-spec*$(S) \subseteq \Sigma^*_{spec}$ is the relaxation of S guided by Q. A sequence h is in *Q-spec*(S), if there exists a sequence $s \in S$ and h', such that h is a prefix of h' and for any $d \in$ *domain*(Q), *distance*$(h' \uparrow_d, s \uparrow_d) \leq Q(d)$. Each Q-quasi sequential specification *Q-spec*(S) is also prefix closed. Given a quasi-linearization factor Q, a history h is *Q-quasi-linearizable* with respect to a sequential specification S, if there exists a sequential history $s = \mathit{call}(i_1, m_1, a_1) \cdot \mathit{return}(i_1, m_1, b_1) \cdot \ldots \cdot \mathit{call}(i_u, m_u, a_u) \cdot \mathit{return}(i_u, m_u, b_u)$ (referred to as *the quasi-linearization of* h), such that $m_1(a_1, b_1) \cdot \ldots \cdot m_u(a_u, b_u) \in$ *Q-spec*(S), h can be extended (by appending zero or more return actions) to a history h', and

- *complete*(h') is equivalent to s.
- For each operation e_1, e_2 of h, if *res*(e_1) precedes *inv*(e_2) in h, then this also holds in s.

Definition 2 (*Q-quasi-linearizability* [2]). *A library $\mathcal{L}$ is Q-quasi-linearizable with respect to a sequential specification S for n processes, if each history of $[\![\mathcal{L}, n]\!]_{cs}$ is Q-quasi-linearizable with respect to S.*

Let Q_{lin} be a quasi-linearization factor, whose domain contains only the element Σ_{spec}. Specially, it maps Σ_{spec} to 0. It is easy to see that Q_{lin}-quasi-linearizability is equivalent to the standard notion of linearizability. One feature of Q-quasi-linearizability is that it allows specifying different deviations to different subsets of Σ_{spec}. For example, a Q-quasi-linearizable queue may have accurate dequeue operations but inaccurate enqueue operations that can bypass at most k preceding enqueue operations [2]. This feature captures the flexibility of possible relaxations, but it also leads to the undecidability result that will be proved in the later section. In the rest of this paper, Q-quasi-linearizability and a Q-quasi-sequential specification are abbreviated as quasi-linearizability and a quasi-sequential specification, respectively, if the context is clear. Given a library $\mathcal{L}$, a sequential specification S, a quasi-linearization factor Q and a positive integer n, the decision problem of quasi-linearizability is to determine whether $\mathcal{L}$ is Q-quasi-linearizable with respect to S for n processes.

4 Undecidability of Quasi-Linearizability

As the main result of this paper, we show in this section that the quasi-linearizability problem is undecidable with respect to a regular sequential specification for a bounded number of processes. We first reduce the k-Z decision problem of a k-counter machine to a linearizability problem of a specific concurrent library for one process. Then, our main undecidability result follows from the correspondence between the linearizability problem and the quasi-linearizability problem of the same library for one process. Since the quasi-linearizability problem of the specific library for one process is equivalent to that for multiple processes, the k-Z decision problem is finally reduced to the quasi-linearizability problem for a bounded number of processes.

4.1 *k*-Counter Machine

The control of a k-counter machine is a finite state automaton, whose alphabet is made up of increment, decrement and test operations to each counter [3]. Let $\Sigma_{ck} = \{I_j, D_j, Z_j | 1 \leq j \leq k\}$ be the set of operations for each counter, where I_j, D_j and Z_j respectively represent the operations for increasing the value of counter j by 1, decreasing the value of counter j by 1 and testing whether the value of counter j is 0. A k-counter machine is a finite state automaton CM $= (Q_{cm}, q_{icm}, F_{cm}, \Sigma_{cm}, \rightarrow_{cm})$, where Q_{cm} is a set of control states, q_{icm} is the initial state, F_{cm} is a set of final states, $\Sigma_{cm} = \Sigma_{ck}$ is a set of transition labels and $\rightarrow_{cm} \subseteq Q_{cm} \times \Sigma_{cm} \times Q_{cm}$ is a transition relation.

Given a finite sequence $l \in \Sigma_{ck}^*$, we use $c_{l,j} = |l \uparrow_{\{I_j\}}| - |l \uparrow_{\{D_j\}}|$ to denote the difference between the numbers of increments and decrements to counter j. We say that a sequence $l \in \Sigma_{ck}^*$ is *admitted*, if for each j and each prefix $l' \cdot Z_j$ $(1 \leq j \leq k)$ of l, $c_{l',j} = 0$. Otherwise, this sequence is *unadmitted*. A k-counter machine CM accepts a finite sequence $\alpha_1 \cdot \ldots \cdot \alpha_m$, if there exists states $q_1, \ldots, q_m$, such that $q_m \in F_{cm}$ and $q_{icm} \xrightarrow{\alpha_1}_{cm} q_1 \xrightarrow{\alpha_2}_{cm} \ldots \xrightarrow{\alpha_m}_{cm} q_m$. Let $lang(CM)$ denotes the language of CM, that is, $lang(CM)$ contains exactly all the sequences that are accepted by CM. The k-Z decision problem is to determine, for a given k-counter machine CM, whether there exists an admitted sequence $l \in lang(CM)$. According to [3], the k-Z decision problem is undecidable, as stated in the following lemma.

Lemma 1 (Undecidability [3]). *The k-Z decision problem is undecidable for* $k \geq 3$.

4.2 Libraries for Prefix Closed Regular Languages

For a finite state automaton R that accepts a prefix closed regular language and whose states are all final states, we can simulate R by the behavior of a specific library $\mathcal{L}_R$ that is constructed based on R.

Formally, given a finite state automaton $R = (Q_r, q_{ir}, F_r, \Sigma_r, \rightarrow_r)$, where Q_r is a set of states, q_{ir} is the initial state, $F_r = Q_r$ is a set of final states, Σ_r is a set of transition labels and $\rightarrow_r$ is a transition relation, the library $\mathcal{L}_R$ is constructed as follows:

- the data domain of $\mathcal{L}_R$ is $Q_r \cup \Sigma_r \cup \{0, 1, true, false, regV_{init}\}$;
- $\mathcal{L}_R$ has two private memory locations *curState* and *flag*. *curState* is used to record the current control state of R, while *flag* is used to ensure mutual exclusion accesses. The initial value of *curState* is q_{ir} and the initial value of *flag* is 0;
- $\mathcal{L}_R$ has one method M, of which the pseudo-code is shown in Method 1. The *if* and *while* statements used in the pseudo-code can be easily implemented with the *assume* commands and other commands in $\mathcal{L}_R$.

The critical section of method M is the region from a successful *cas* command at Line 1 to the $flag = 0$ command at Line 5. M first waits until it enters the critical section (Lines 1–2). If there exists one step of transition in R starting from *curState* (Line 3), M changes the value of *curState* according to this transition, leaves the critical section and returns the transition label (Lines 4–6). Otherwise, M is blocked (Lines 7–9).

The pseudo-code in Lines 1–2, where a *cas* operation is used, together with the pseudo-code at Line 5, ensure the mutual exclusion between invocations of this method.[1] We use $lang(R)$ to denote the language of R.

[1] Except the *cas* operation, other operations, such as filter lock [10] can also be used herein to ensure mutual exclusion.

Method 1. M

```
Input: an arbitrary argument
Output: transition label for one step in $R$
1 while $cas(flag, 0, 1)$ fails do
2     ;
3 if there exists some $q, \alpha$, such that $curState \xrightarrow{\alpha}_r q$ then
4     $curState = q$;
5     $flag = 0$;
6     return $\alpha$;
7 else
8     while true do
9         ;
```

4.3 Reducing a *k-Z* Decision Problem to a Linearizability Problem

In this subsection we show that the *k-Z* decision problem of a *k*-counter machine can be reduced to the linearizability problem of a specific library for one process with respect to a non-regular sequential specification. This reduction is achieved with the aid of a language inclusion problem.

It is not hard to see that the set of all unadmitted sequences is far beyond the scope of regular languages. Fortunately, according to [3], there is a regular set of "templates" corresponding to the set of all unadmitted sequences. For instance, the unadmitted sequence $l = I_1 \cdot I_2 \cdot I_1 \cdot I_1 \cdot D_1 \cdot D_1 \cdot Z_1 \cdot D_2 \cdot Z_2$ contains a minimal unadmitted prefix $l' = I_1 \cdot I_2 \cdot I_1 \cdot I_1 \cdot D_1 \cdot D_1 \cdot Z_1$. Let l'' be the projection of l' to counter 1, i.e., $l'' = I_1 \cdot I_1 \cdot I_1 \cdot D_1 \cdot D_1 \cdot Z_1$. It can be seen that l'' is also unadmitted. The template for l can be constructed as the concatenation of two parts. The first part, $I_1 \cdot D_1 \cdot I_1 \cdot D_1 \cdot I_1 \cdot Z_1$, is constructed from l'' by swapping the locations of I_1 and D_1. Such swapping tries to pair as many matching I_1 and D_1 as possible in the beginning of the sequence, while ensuring that $I_1 s$ and $D_1 s$ do not cross Z_1. The second part, $I_2 \cdot D_2 \cdot Z_2$, is the rest contents of l.

Formally, the regular set of templates for the set of all unadmitted sequences of k-counter machines is

$$\bigcup_{j=1}^{k} (((I_j \cdot D_j)^* \cdot Z_j^*)^* \cdot (I_j \cdot D_j)^* \cdot (I_j^+ + D_j^+) \cdot Z_j \cdot \Sigma_{ck}^*)$$

where $\bigcup_{j=1}^{k} \mathcal{A}_j = \mathcal{A}_1 + \ldots + \mathcal{A}_k$. Since this template set will be used later to construct prefix closed sequential specifications, we further extended it to a prefix closed regular language

$$k\text{-}US = \Sigma_{ck}^* + \bigcup_{j=1}^{k} (((I_j \cdot D_j)^* \cdot Z_j^*)^* \cdot (I_j \cdot D_j)^* \cdot (I_j^+ + D_j^+) \cdot Z_j \cdot \Sigma_{ck}^* \cdot end)$$

where *'US'* represents that this set can be considered as the specification for all unadmitted sequences.

The set $Q_k = \{\{I_j, Z_j\}, \{D_j, Z_j\} | 1 \leq j \leq k\} \cup \{\{end, act\} | act \in \Sigma_{ck}\}$ will be used to guide the relaxation to k-US, which ensures that each increment and decrement operation can not cross a test operation to the same counter, and each operation in Σ_{ck} can not cross the *end* symbol. The relaxation to k-US guided by Q_k is the set Q_k-$set(k\text{-}US) = \{l | \exists l' \in k\text{-}US, \forall d \in Q_k, l \uparrow_d = l' \uparrow_d\}$. Each element l of Q_k-$set(k\text{-}US)$ is either a sequence in Σ_{ck}^* or the concatenation of an unadmitted sequence and the *end* symbol. Moreover, for each unadmitted sequence l, $l \cdot end$ is a member of Q_k-$set(k\text{-}US)$.

Although the language of a k-counter machine may not be prefix closed, we can construct a prefix closed regular language for a k-counter machine. Later, a specific library will be constructed from this language (automaton) and used as a bridge to connect a language inclusion problem and a linearizability problem. Given a k-counter machine $CM = (Q_{cm}, q_{icm}, F_{cm}, \Sigma_{cm}, \rightarrow_{cm})$, we construct the finite state automaton $R_{CM} = (Q, q_i, F, \Sigma, \rightarrow)$ as follows:

- $Q = Q_{cm} \cup \{q_{end}\}$ is the set of states, where q_{end} is a new state not in Q_{cm}.
- $q_i = q_{icm}$ is the initial state.
- $F = Q$ is the set of final states.
- $\Sigma = \Sigma_{cm} \cup \{end\}$ is the set of transition labels, where *end* is a new transition label not in Σ_{cm}.
- $\rightarrow = \rightarrow_{cm} \cup \{(q_f, end, q_{end}) | q_f \in F_{cm}\}$ is the transition relation.

Given a set S of sequences, let $S \cdot a = \{l | \exists l' \in S, l = l' \cdot a\}$ denote the set of concatenations of sequences in S and a symbol a. It is not hard to see that the language of R_{CM}, denoted as $lang(R_{CM})$, is the union of

- $lang(CM) \cdot end$,
- $\{\alpha_1 \cdot \ldots \cdot \alpha_m | \exists q_1, \ldots, q_m, q_{icm} \xrightarrow{\alpha_1}_{cm} q_1 \xrightarrow{\alpha_2}_{cm} q_2 \ldots \xrightarrow{\alpha_m}_{cm} q_m\}$.

It can be seen that $lang(R_{CM})$ is a prefix closed language.

Figure 2 shows an example R_{CM} that is generated from a k-counter machine CM. The counter machine CM has four states: q_1, q_2, q_3, and q_4, the last two of which are final states. During the construction of this R_{CM}, we add a new state q_{end} and two additional transitions from q_3 and q_4 to q_{end}, and make all states as final states.

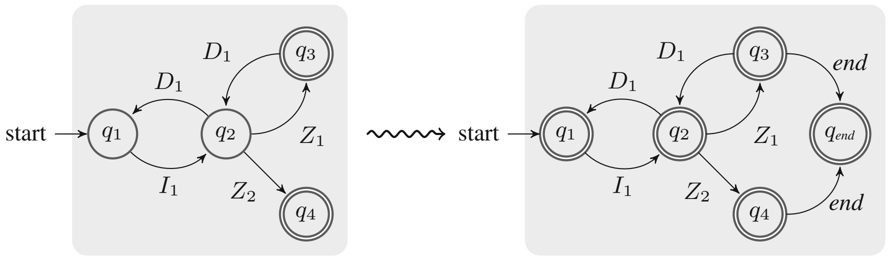

Fig. 2. Generation of R_{CM} from a counter machine CM

The following lemma reduces the *k-Z* decision problem of a *k*-counter machine *CM* to the language inclusion problem between $lang(R_{CM})$ and $Q_k\text{-}set(k\text{-}US)$.

Lemma 2. *Given a k-counter machine CM, all the sequences in lang(CM) are not admitted if and only if* $lang(R_{CM}) \subseteq Q_k\text{-}set(k\text{-}US)$.

Proof. To prove the *if* direction, for each sequence $l \in lang(CM)$, it is easy to find that $l \cdot end$ is in $lang(R_{CM})$. By assumption, $l \cdot end$ also belongs to $Q_k\text{-}set(k\text{-}US)$. It is obvious that each sequence ending with an *end* symbol in $Q_k\text{-}set(k\text{-}US)$ is a concatenation of an unadmitted sequence and an *end* symbol, so l is not admitted.

To prove the *only if* direction, given a counter machine $CM = (Q_{cm}, q_{icm}, F_{cm}, \Sigma_{cm}, \rightarrow_{cm})$, then for each sequence $l = \alpha_1 \cdot \ldots \cdot \alpha_m \in lang(R_{CM})$, we can observe that

- If $\alpha_m \neq end$ and there exists $q_1, \ldots, q_m$, such that $q_{icm} \xrightarrow{\alpha_1}_{cm} q_1 \xrightarrow{\alpha_2}_{cm} \ldots \xrightarrow{\alpha_m}_{cm} q_m$, then $l \in \Sigma_{ck}^*$. Thus it is an element of *k-US*, and furthermore an element of $Q_k\text{-}set(k\text{-}US)$.
- If $l \in lang(CM) \cdot end$, then $\alpha_1 \cdot \ldots \cdot \alpha_{m\text{-}1} \in lang(CM)$ and $\alpha_m = end$. By assumption $\alpha_1 \cdot \ldots \cdot \alpha_{m\text{-}1}$ is an unadmitted sequence, and it is easy to see that $l \in Q_k\text{-}set(k\text{-}US)$.

In both situations, $l \in Q_k\text{-}set(k\text{-}US)$. This completes the proof. □

Given a set T of sequences over $\mathcal{D}$, we can lift it to a sequential specification $MSeqSpec(T) = \{M(\_, b_1) \cdot \ldots \cdot M(\_, b_u) | b_1 \cdot \ldots \cdot b_u \in T\}$ for the libraries constructed in Subsect. 4.2. Given a set $S \subseteq \Sigma_{spec}^*$, we use $seqHis(S, n)$ to denote all the sequential histories that are generated from a sequence in S by substituting each $m(a, b)$ in the sequence with a pair of a call action $call(i, m, a)$ and its matching return action $return(i, m, b)$ for some i, where $1 \leq i \leq n$. The following lemma reduces a language inclusion problem to a linearizability problem for one process.

Lemma 3. *Given a sequential specification S and a finite state automaton R that accepts a prefix closed language,* $\mathcal{L}_R$ *is linearizable with respect to S for one process if and only if* $MSeqSpec(lang(R)) \subseteq S$.

Proof. The *if* direction is proved as follows. For each history $h \in history$ $([\![\mathcal{L}_R, 1]\!]_{cs})$, since the critical sections of h are constructed according to transitions of R, as well as the fact that each return action returns a transition label of a critical section, we have that $h \in SeqHis(MSeqSpec(lang(R)), 1)$. By assumption, it can be seen that $h \in SeqHis(S, 1)$. Because h itself is a sequential history, it is obvious that h is linearizable with respect to S.

The *only if* direction is proved by contradiction. Assume that L_R is linearizable with respect to S for one process but $MSeqSpec(lang(R))$ is not a subset of S. Therefore there exists a sequence $l = M(a_1, b_1) \cdot \ldots \cdot M(a_m, b_m) \in MSeqSpec(lang(R)) - S$. Let sequential history $h' = call(1, M, a_1) \cdot return(1, M, b_1) \cdot \ldots \cdot call(1, M, a_m) \cdot return(1, M, b_m)$. Because $b_1 \cdot \ldots \cdot b_m \in lang(R)$ and each return value of M is the transition label of one step transition of R, sequential history h' is a history of $[\![\mathcal{L}_R, 1]\!]_{cs}$ and h' can not be linearizable with respect to S. But this contradicts the fact that $\mathcal{L}_R$ is linearizable with respect to S. □

Given a k-counter machine *CM*, with Lemmas 2 and 3, we can reduce the problem of whether all the sequences in *lang*(*CM*) are unadmitted to that of whether the specific library $\mathcal{L}_{R_{CM}}$ is linearizable with respect to the sequential specification *MSeqSpec*(Q_k-*set*(k-*US*)) for one process.

4.4 Undecidability of Quasi-Linearizability

In this subsection we reduce the problem of whether $\mathcal{L}_{R_{CM}}$ is linearizable with respect to the sequential specification *MSeqSpec*(Q_k-*set*(k-*US*)) to the problem of whether $\mathcal{L}_{R_{CM}}$ is quasi-linearizable with respect to a regular sequential specification.

Recall that Q_k-*set*(k-*US*) is the relaxation of k-*US* guided by Q_k, and such relaxation does not permit I_j and D_j to go across Z_j for each counter j, and does not permit operations in Σ_{ck} to cross *end*. Correspondingly, a relaxed specification can be obtained by relaxing *MSeqSpec*(k-*US*) in a similar way as the Q_k relaxation does to k-*US*. The following lemma shows that the sequential specification of the above linearizability problem can be precisely reproduced by certain Q'_k-quasi-sequential specification, which is generated by relaxing *MSeqSpec*(k-*US*) with the quasi-linearization factor Q'_k. In this way, the above linearizability problem can be reduced to a quasi-linearizability problem. The Quasi-linearization factor Q'_k maps every element in D_k to 0, where D_k is the union of the following sets:

- $\{M(a, I_j), M(b, Z_j) | a, b \in \mathcal{D}\}$, where $1 \leq j \leq k$.
- $\{M(a, D_j), M(b, Z_j) |\ a, b \in \mathcal{D}\}$, where $1 \leq j \leq k$.
- $\{M(a, \mathit{end}), M(b, \mathit{act}) | a, b \in \mathcal{D}\}$, where $\mathit{act} \in \Sigma_{ck}$.

Lemma 4. *A library $\mathcal{L}$ is linearizable with respect to MSeqSpec(Q_k-set(k-US)) for one process if and only if $\mathcal{L}$ is Q'_k-quasi-linearizable with respect to MSeqSpec(k- US) for one process.*

Proof. The *if* direction is proved as follows. For each history $h \in \mathit{history}$ $([\![\mathcal{L}, 1]\!]_{cs})$, assume its quasi-linearization $s = \mathit{call}(1, M, a_1) \cdot \mathit{return}(1, M, b_1) \cdot \ldots \cdot \mathit{call}(1, M, a_m) \cdot \mathit{return}(1, M, b_m)$. We can construct a sequence $l_1 = M(a_1, b_1) \cdot \ldots \cdot M(a_m, b_m)$ from s. By definition of quasi-linearizability, $l_1 \in Q'_k$-*spec*(*MSeqSpec*(k-*US*)). There exist sequences $l_2 = M(a_{m+1}, b_{m+1}) \cdot \ldots \cdot M(a_{m+u}, b_{m+u})$ and $l_3 = M(a'_1, b'_1) \cdot \ldots \cdot M(a'_{m+u}, b'_{m+u})$.

- $l_3 \in$ *MSeqSpec*(k-*US*).
- $\forall d \in D_k$, $\mathit{distance}((l_1 \cdot l_2) \uparrow_d, l_3 \uparrow_d) = 0$.

Thus we can find that $\forall d' \in Q_k$, $(b_1 \cdot \ldots \cdot b_m \cdot b_{m+1} \cdot \ldots \cdot b_{m+u}) \uparrow_{d'} = (b'_1 \cdot \ldots \cdot b'_m \cdot b'_{m+1} \cdot \ldots \cdot b'_{m+u}) \uparrow_{d'}$. Based on this fact, we immediately obtain that $b_1 \cdot \ldots \cdot b_m \cdot b_{m+1} \cdot \ldots \cdot b_{m+u} \in Q_k$-*set*($k$-*US*), thus the sequence $b_1 \cdot \ldots \cdot b_m \in$ *Prefix*(Q_k-*set*(k-*US*)). Because Q_k-*set*(k-*US*) is prefix closed, $b_1 \cdot \ldots \cdot b_m \in Q_k$-*set*($k$-*US*). Therefore, s belongs to *SeqHis*(*MSeqSpec*(Q_k-*set*(k-*US*)), 1). It is obvious that h is linearizable with respect to *MSeqSpec*(Q_k-*set*(k-*US*)).

The *only if* direction can be proved in a similar way. □

The following theorem shows that the quasi-linearizability problem is undecidable with respect to a regular sequential specification for one process.

Theorem 1. *Given a library $\mathcal{L}$, a regular sequential specification S and a quasi-linearization factor Q, it is undecidable whether $\mathcal{L}$ is Q-quasi-linearizable with respect to S for one process.*

Proof. Given a k-counter machine *CM*, by Lemma 2, the problem of whether all sequences in *lang*(*CM*) are unadmitted can be reduced to the language inclusion problem between *lang*(R_{CM}) and Q_k-*set*(k-*US*). It can be seen that the latter problem is equivalent to the language inclusion problem between *MSeqSpec*(*lang*(R_{CM})) and *MSeqSpec*(Q_k-*set*(k-*US*)). By Lemma 3, it can be reduced to the problem of whether $\mathcal{L}_{R_{CM}}$ is linearizable with respect to *MSeqSpec*(Q_k-*set*(k-*US*)) for one process. By Lemma 4, it can be further reduced to the problem of whether $\mathcal{L}_{R_{CM}}$ is Q'_k-quasi-linearizable with respect to *MSeqSpec*(k-*US*) for one process. It is easy to see that *MSeqSpec*(k-*US*) is regular. Recall that by Lemma 1 the k-Z decision problem is undecidable for $k \geq 3$. This completes the proof of this theorem. $\square$

The specific library $\mathcal{L}_R$ has the following property: for each positive integer n and history $h \in$ *history*($[\![\mathcal{L}_R, n]\!]_{cs}$), we can construct a sequential history of *history*($[\![\mathcal{L}_R, 1\ ]\!]_{cs}$) according to the critical section accesses in h, in the way as shown in Fig. 3. Assume history $h \in$ *history*($[\![\mathcal{L}_R, n]\!]$) contains call actions c_i and return actions r_i, where $1 \leq i \leq 3$. In Fig. 3, the time axes run from left to right, while each method is associated with a line interval. Additionally, the time intervals of the critical section accesses are marked with the shadow regions in Fig. 3. It is not hard to see that if we change the positions of each pair of a call action and its matching return action to the nearest time point before and after the corresponding critical section accesses, we get a sequential history $h' \in$ *history*($[\![\mathcal{L}_R, n]\!]_{cs}$). Since h' is sequential, we can construct another sequential history $h'' \in$ *history*($[\![\mathcal{L}_R, 1]\!]_{cs}$) of a single process by moving all actions of h' to this process. It is obvious that h'' contains exactly all the actions of h' and preserves exactly the sequential order of all the actions of h'.

Since the specific library $\mathcal{L}_R$ runs in a sequential way, the following lemma shows that the quasi-linearizability problem of $\mathcal{L}_R$ for one process can be reduced to the quasi-linearizability problem of the same library for more than one process.

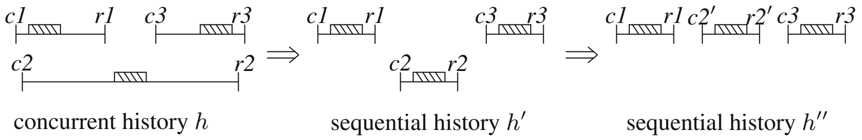

Fig. 3. Construction of a sequential history of a unique process

Lemma 5. *Given a sequential specification S, a finite state automaton R that accepts a prefix closed language, a quasi-linearization factor Q and a positive* *integer $n > 1$, $\mathcal{L}_R$ is Q-quasi-linearizable with respect to S for n processes if and only if $\mathcal{L}_R$ is Q-quasi-linearizable with respect to S for one process.*

Proof. The *only if* direction is obvious since $\mathit{history}(\llbracket \mathcal{L}_R, 1 \rrbracket_{cs}) \subseteq \mathit{history}(\llbracket \mathcal{L}_R, n \rrbracket_{cs})$.

The *if* direction is proved by contradiction. Assume that $\mathcal{L}_R$ is Q-quasi-linearizable with respect to S for one process, but not for n processes. Then there must be a history $h \in \mathit{history}(\llbracket \mathcal{L}_R, n \rrbracket_{cs})$ such that h is not Q-quasi-linearizable with respect to S.

As shown in Fig. 3, we can construct a sequential history h' of $\llbracket \mathcal{L}_R, n \rrbracket_{cs}$ from h by changing the positions of each pair of call and return actions to the nearest time point before and after the corresponding critical section. During this process, it is necessary to add some return actions for the case when a method has completed its critical section but not returned yet, it is also necessary to remove some call actions for the case when a method has not entered its critical section. By assumption, it can be seen that h' is not Q-quasi-linearizable with respect to S.

Similarly, we can also construct a sequential history h'' of $\llbracket \mathcal{L}_R, 1 \rrbracket_{cs}$ from h' by moving all actions of h' to the unique process. By assumption, it can be seen that h'' is not Q-quasi-linearizable with respect to S. But this contradicts the assumption that $\mathcal{L}_R$ is Q-quasi-linearizable with respect to S for one process. □

The following theorem states that the quasi-linearizability problem is undecidable with respect to a regular sequential specification for a bounded number $n \geq 1$ of processes. This is a direct consequence of Theorem 1 and Lemma 5.

Theorem 2. *Given a library $\mathcal{L}$, a regular sequential specification S, a quasi-linearization factor Q and a positive integer $n \geq 1$, it is undecidable whether $\mathcal{L}$ is Q-quasi-linearizable with respect to S for n processes.*

5 Conclusion and Future Work

We show in this paper that the quasi-linearizability problem with respect to a regular sequential specification is undecidable for a bounded number of processes. This is essentially proved by reduction from the k-Z problem of a k-counter machine, a known undecidable problem. We prove that the k-Z problem can be reduced to a language inclusion problem, which can be further reduced to a linearizability problem of a specific library for just one process. The library is constructed from the k-counter machine and can simulate its behavior. Then, this linearizability problem can be reduced to a quasi-linearizability problem with respect to a regular sequential specification for a bounded number of processes.

Note that although the sequential specification of the quasi-linearizability problem is regular, its quasi-sequential counterpart is non-regular and rather complex. Thus, a quasi-linearizability problem with respect to a regular sequential specification is equivalent to a linearizability problem with respect to a non-regular sequential specification. Since quasi-linearizability is undecidable for just one process with respect to a regular sequential specification, the undecidability of quasi-linearizability is not resulted from the interactions between processes, but from the fact that no element in D_k is equal to Σ_{spec}. Actually, if the domain of quasi-linearization factor contains only one element Σ_{spec}, then quasi-linearizability problem with respect to regular sequential specification is decidable for a bounded number of processes, as shown in [1]. Therefore, we can conclude that the undecidability of quasi-linearizability is inherent from its flexibility in quantitative relaxation.

Quasi-linearizability has been deprecated since its syntactic distance definition has been demonstrated broken [9]. Compared to the syntactic relaxation in [2], the relaxation in [9] is based on library semantics, and it offers an accurate and efficient way to relax linearizability. We conjecture that quasi-linearizability is a special case of the quantitative relaxation framework presented in [9], of which the decidability is still unknown. Thus, our undecidability work for quasi-linearizability can be used as a gateway to this open problem. As future work we would like to investigate the decision problem of this quantitative relaxation framework for a bounded number of processes.

Acknowledgments. The authors of this paper would like to thank anonymous reviewers for pointing out the connection between dependence relations in [3] and quasi-linearization factors in [2].

References

1. Adhikari, K., Street, J., Wang, C., Liu, Y., Zhang, S.J.: Verifying a quantitative relaxation of linearizability via refinement. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 24–42. Springer, Heidelberg (2013)
2. Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: relaxed consistency for improved concurrency. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 395–410. Springer, Heidelberg (2010)
3. Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. In: LICS 1996, pp. 219–228. IEEE Computer Society (1996)
4. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. ACM **41**(5), 1020–1048 (1994)
5. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 290–309. Springer, Heidelberg (2013)
6. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Rajamani, S.K., Walker, D. (eds.) POPL 2015, pp. 651–662. ACM (2015)
7. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the TSO memory model. In: Seidl, H. (ed.) Programming Languages and Systems. LNCS, vol. 7211, pp. 87–107. Springer, Heidelberg (2012)
8. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010)

9. Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013, pp. 317–328. ACM (2013)
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2008)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
12. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Trans. Comput. **28**(9), 690–691 (1979)
13. Zhang, L., Chattopadhyay, A., Wang, C.: Round-up: Runtime checking quasi linearizability of concurrent data structures. In: Denney, E., Bultan, T., Zeller, A. (eds.) ASE 2013, pp. 4–14. IEEE (2013)