ISCAS-SKLCS-21-01 July, 2021

中国科学院软件研究所
计算机科学国家重点实验室
技术报告

Bounded Linearizability on TSO Memory Model is Decidable

by

Chao Wang, Yi Lv, Peng Wu, Qiaowen Jia

State key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing 100190. China

Bounded Linearizability on TSO Memory Model is Decidable

Chao Wang[1], Yi Lv[2], Peng Wu[2], and Qiaowen Jia[2]

[1] Southwest University, China
[2] State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences

Abstract. TSO-to-TSO linearizability is a typical variants of linearizability for concurrent data structures on the Total Store Order (TSO) memory model. In this paper we propose the notion of k-bounded TSO-to-TSO linearizability, a subclass of TSO-to-TSO linearizability that concerns only extended histories of bounded length. These subclasses are non-trivial in that they does not restrict the number of write, flush and *cas* (compare-and-swap) actions, nor the size of a store buffer, to be bounded.
We prove that the decision problem of k-bounded TSO-to-TSO linearizability is decidable for a bounded number of processes. In the semantics of (k-bounded) TSO-to-TSO linearizability, a call or return action influences control state and store buffer at the same time. We divide a "composed" call action into a "pure" call action and an action for store buffer. Then, a "pure" call action is marked with a specific *cas* action by introduce a observer process and modify the memory model to make sure a specific *cas* action is "bind" to call action. The role for store buffer of a "composed" call action is transformed into a write action. Then, we need only to concentrate on actions of memory models (read, write, flush, *cas*) and we reduce the k-bounded TSO-to-TSO linearizability problem to several control state reachability problems of lossy channel machines, which is already known to be decidable.
In a similar manner, we can conclude that the decision problems of k-bounded TSO-to-SC linearizability and TSO linearizability are also decidable for a bounded number of processes. Thus, all bounded linearizability on TSO are decidable.

1 Introduction

High performance libraries of concurrent data structures, such as *java.util.concurrent* for Java and *std::thread* for C++11, have been widely used in concurrent programs to take advantage of multi-core architectures. It is important but notoriously difficult to ensure that concurrent data structures are designed and implemented correctly. *Linearizability* [1] is accepted as a *de facto* correctness condition for a concurrent data structure with respect to its sequential specification on the sequential consistency (SC) memory model [2].

However, modern multiprocessors (e.g., x86 [3], POWER [4]) and programming languages (e.g., C/C++ [5], Java [6]) do not comply with the SC memory model. As a matter of fact, they provide *relaxed memory models*, which allow subtle behaviors due to hardware or compiler optimization. For instance, in a multiprocessor system implementing the total store order (TSO) memory model [3], each processor is equipped with an FIFO store buffer. Any write action performed by a processor is put into its local store buffer first and can then be flushed into the main memory at any time. The TSO memory model requires that all processes in a concurrent system observe the same order of update actions (flush and *cas* actions), which is referred to as a total store order.

The notion of linearizability has been extended for relaxed memory models, e.g., *TSO-to-TSO linearizability* [7] and *TSO-to-SC linearizability* [8] for the TSO memory model and two variants of linearizability [5] for the C++ memory model. These notions generalize the original one by relating concurrent data structures with their abstract implementations, in the way as shown in [9] for the SC memory model.

It is well known that the linearizability of a concurrent library on the SC memory model is decidable for a bounded number of processes [10], but undecidable for an unbounded number of processes [11]. However, to our knowledge, there are only a few decidability results about linearizability on relaxed memory models. We have recently proved that the decision problem of *TSO-to-TSO linearizability* is undecidable for a bounded number of processes [12,13]. But the decision problem of TSO-to-SC linearizability still remains open for a bounded number of processes.

TSO-to-TSO linearizability relates a concurrent data structure running on the TSO memory model to its abstract implementation running also on the TSO memory model. When method invocation (call) or responses (return) happen, a marker is put into local store buffer first, and its flushing will launches a flushCall or flushReturn action. TSO-to-TSO linearizability use extended histories to represent the behaviors of concurrent data structures, which are sequences of call, return, flushCall and flushReturn actions. We propose a decidable subclass of TSO-to-TSO linearizability, which is referred to as *k-bounded TSO-to-TSO linearizability*. It concerns only k-extended histories, which are extended histories with less or equal than k actions. Note that the k-boundedness on the number of call, return, flushCall and flushReturn actions does not necessarily restrict the behaviors of a concurrent program to be finite-state.

Since the set of all k-extended histories is bounded, if we can effectively obtain the set of k-extended histories of a concurrent data structure for bounded number of processes, then we can decide k-bounded TSO-to-TSO linearizability. Inspired by [14], we consider to reduce the the problem of whether a concurrent data structure has a specific k-extended history to several control state reachability problem of a lossy channel machine, which is known decidable [14]. However, the reduction method in [14] does not directly apply to k-bounded TSO-to-TSO linearizability. This is because that method in [14] only consider internal actions or actions of memory model, such as read, write or *cas* (compare-and-swap) actions. As contrast, call and return actions in TSO-to-TSO linearizability are much different. A call action in TSO-to-TSO linearizability has two roles that are taken simultaneously: The first role is to change the control state of the calling process, and the second role is to insert a call marker into store buffer of the calling process. The central idea of our work is to “transform” the two roles of a call actions into a write action and a *cas* action, which belongs to the memory model (the case of return action is similar).

It is hard to investigate decidability of k-bounded TSO-to-TSO linearizability with such "composed" call and return actions. Therefore, we separate a "composed" call action into a "pure" call action that only change the control state of the calling process, and a immediately followed write action that inserts a call marker into buffer. To make this, we introduces a new memory location z_f and modify the operational semantics of TSO-to-TSO linearizability as follows: (1) Call and return actions are "pure" now, or we can say, they do not put anything into buffer. (2) We additionally add a write action of z_f just after each call and return actions, and write call or return markers, respectively. (3) When the item of z_f is flushed out of buffer, we launch a flushCall or flushReturn action. In this way, the second role of a "composed" call is mimicked by the insertion of z_f and the "TSO mechanism".

It is also hard to deal with "pure" call and return action, since they are to some extend "beyond" TSO memory model and seems not compatible with TSO memory model. Let us explain this. A call action takes effect as soon as it happens. As contrast, on TSO, write actions can be seen by other process only when they are flushed. This implies that, if a process want to notice another process by writing something into buffer, then this message can be seen by other process only when it is flushed. To make it worse, the order of call and return action may be different from the order of how items in buffer are flushed. Since the content of buffer may influence future execution, a process may not be able to notice other process of call or return actions by using *cas* actions, since a *cas* action will clear the buffer. Therefore, it is hard to make a concurrent system to know the order of call and return actions without influence its future execution on TSO. To solve this problem, we modify TSO memory model and make call and return actions visible to concurrent system by "bind" each call or return action with a specific *cas* action. To be exact, we introduces a new memory location z_w, a new "observer" process, which keeps launching the specific *cas* actions of z_w nondeterministically, and modify the TSO memory model, in a way where each call or return action must be bind (immediately before) to a specific *cas* action of the observer process. Or we can say, the concurrent system can not proceed as soon as a *cas* action of observer process is not immediately followed by the corresponding call or return action. For each execution of this modified operational semantics, the order of call and return actions are same as the order of *cas* actions in observer process. In this way, we mimic the "pure" call and return actions with *cas* actions of observer process. For extended history *eh* in TSO memory model, there is a trace t in the modified TSO model, where t has extended history *eh* and all items putted into buffer in t has been flushed. Such a trace is called a marked witness of the extended history. The existence of extended history in TSO memory model is equivalent to the existence of marked witness of the extended history in the modified TSO model. Since we have "transformed" the "composed" call and return actions with write and *cas* actions with memory model modified, the method in [14] can be applied to investigate the existence of marked witness.

A lossy channel machine M_i^k $(1 \leq i \leq n{+}1)$ is then constructed, which simulates the k-bounded behaviors of the concurrent system from the perspective of each process P_i, and ensure that each specific *cas* of the observer process is immediately followed by a corresponding call or return action. M_i^k also nondeterministically guesses actions of other process, and contains only one channel to store the pending written items according to the total store orders under the original concurrent system. Thus, the existence of a marked witness of an extended history can be reduced to a control state reachability problem between one pair of specific states of the product of $M_1^{k\text{-}w}, \ldots, M_{n+1}^{k\text{-}w}$. Each $M_i^{k\text{-}w}$ is resulted from M_i^k by replacing its all but write, *cas*, flushCall and flushReturn (flush of z_f) transitions with internal transitions. The reduction is achieved by requiring that each written item in a channel contains a run-time snapshot of the memory, while always keeping bounded the amount of information that needs to be stored as in a perfect channel. With these specialized lossy channels, missing some intermediate channel contents would not break the reachability between control states under perfect channels. Since the number of pairs of specific states is finite, we can reduce the existence of a marked witness of an extended history to a finite number times of control state reachability problem of lossy channel machines.

Further, we can show that the decision problem of k-bounded TSO-to-TSO linearizability has at least non-primitive recursive complexity. This can be proved by a reduction from a reachability problem of a lossy simple channel machine, which is known to have non-primitive recursive complexity [15]. Inspired by our previous work [13], we generate a template $\mathcal{L}_{(s_1,s_2)}^M$ for simulating transitions from s_1 to s_2, which are two states of lossy simple channel machine M. This concurrent data structure contains two methods M_1 and M_2. We use two buffers of two processes to simulate one channel, where processes P_1 runs M_1 and process P_2 runs M_2. Process P_1 read updates of process P_2, change them according to transition rules of M, and write them into buffer, while process P_2 read updates of process P_1 and write them into buffer. On TSO, between two consecutive read actions, more than one flush actions may happen, but only the latest flush action can be read while the intermediate flush actions can not. Such fact of missing flush actions are used to simulate lossy of channel. Each transition of the lossy simple channel machine is reproduced through the interactions between M_1 and M_2. M_2 never return, while M_1 returns as soon as s_2 is reached. Therefore, the problem of whether s_2 is reachable from s_1 is reduced to checking whether some extended history of this concurrent data structure has a return action of M_1, which can be easily reduced to a 5-bounded TSO-to-TSO linearizability problem.

TSO-to-SC linearizability has been proposed for reasoning about the correctness of a concurrent data structure, which is native to the TSO memory model but is used with a concurrent program that needs to be protected from the relaxed semantics. It relates a concurrent data structure running on the TSO memory model to its abstract implementation running on SC memory model. TSO-to-SC linearizability uses histories, which are sequences of call and return actions, to represent the behaviors of concurrent data structures. We propose a decidable subclass of TSO-to-TSO linearizability, which is referred to as *k-bounded TSO-to-SC linearizability*. It concerns only k-histories, which are histories with less or equal than k actions. Here, the k-boundedness on the number of call and return actions does not necessarily restrict the behaviors of a concurrent program to be finite-state. We prove that k-bounded TSO-to-SC linearizability is also decidable for a bounded number of processes similarly as the proof for k-bounded TSO-to-TSO linearizability.

Apart from TSO-to-TSO linearizability and TSO-to-SC linearizabiltiy, there is another variant of linearizability on TSO called TSO linearizability [16,17], which does not have corresponding abstraction theorem. Essentially, TSO linearizability considers a method to start at its call action and end at its flushReturn action. Or we can say, TSO linearizability use sequences of call and flushReturn actions to represent behaviors of concurrent data structures. To accommodate with TSO linearizability, we generalize history into fifteen variants, where history, extended history and the sequence for TSO linearizability are three variants among them. Then, we prove that we can effectively obtain the set of sequences with bounded length of a concurrent data structure for all fifteen variants of histories. We propose a bounded variants of TSO linearizability called k-bounded TSO linearizability, and prove that k-bounded TSO linearizability is also decidable and has at least nonprimitive recursive complexity.

Related work Efforts have been devoted on verification of linearizability on the SC memory model [10,11,18,19]. Alur *et al.* proved that linearizability is decidable for a bounded number of processes [10], and Bouajjani *et al.* proved that linearizability is undecidable for a unbounded number of processes [11] by a reduction from the reachability problem of a counter machine (which is known to be undecidable).

Atig surveys the verification problem of safety and liveness properties of finite-state programs running under the TSO memory model [20]. However, Relaxed memory models remain a great challenge for linearizability verification. Our previous work [13] revealed the first undecidability result on TSO-to-TSO linearizability for a bounded number of processes. In [13], the trace inclusion problem of a classic-lossy single-channel system, which has been known to be undecidable, was reduced to the TSO-to-TSO linearizability problem. Our work is partly inspired by Atig *et al.* [14], where a state reachability problem of a concurrent system is reduced to a control state reachability problem of a lossy channel machine.

The most closest work to ours is our previous work [12]. However, [12] only consider (k-bounded) TSO-to-SC linearizability, and the lossy channel machine of [12] is used to check whether a concurrent data structure has a history (of bounded length) that violate a regular language. In this paper, we consider the more generalized situation which cover all the variants of linearizability on TSO. We use lossy channel machine to check whether a concurrent data structure has a specific extended history (of bounded length). We further establish the decidability result and complexity of k-bounded TSO-to-TSO linearizability, shows that we can effectively obtain the set of sequences with bounded length of a concurrent data structure for all fifteen kinds of variants of histories. Thus, the result of [12] now is a corollary of our paper.

Paper Outline. Section 2 presents the definitions of concurrent data structures, concurrent systems, and its operational semantics of concurrent systems on TSO and SC memory models. In Section 3, we introduce the definition of TSO-to-TSO linearizability and TSO-to-SC linearizability, and propose the definition of k-bounded TSO-to-TSO linearizability and k-bounded TSO-to-SC linearizability. In Section 4, we give a modified operational semantics that transform the second role of a call (resp., return) actions into a write action. In Section 5, we introduce the definition of perfect/lossy channel machines. In Section 6, we modify the operational semantics and bind each call or return action with a specific *cas* action, and propose the notion of marked witness. In Section 7, we propose the detailed definitions of channel machines for simulating each process while checking existence of a specific extended history. We prove in Section 8 that

k-bounded TSO-to-TSO linearizability is decidable for bounded number of processes. In Section 9, we prove that k-bounded TSO-to-TSO linearizability has non-primitive recursive complexity. In Section 10, we show the related results for (k-bounded) TSO-to-SC linearizability. In Section 10.2, we show that we can effectively obtain the set of sequences with bounded length of a concurrent data structure for all fifteen kinds of variants of histories, and sketch definition and decidability proof of k-bounded TSO linearizability. The article is concluded in Section 11 with future work.

2 Concurrent Systems

In this section, we present the notations of concurrent data structures, client programs, most general clients and concurrent systems. Then, we introduce their operational semantics on the TSO and SC memory model.

2.1 Notations

In general, a finite sequence on an alphabet Σ is denoted $l = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_k$, where $\cdot$ is the concatenation symbol and $\alpha_i \in \Sigma$ for each $1 \leq i \leq k$. Let $|l|$ and $l(i)$ denote the length and the i-th element of l, respectively, i.e., $|l| = k$ and $l(i) = \alpha_i$ for $1 \leq i \leq k$. Let $l \uparrow_\Sigma$ denote the projection of l to Σ. Given a function f, let $f[x : y]$ be the function that is the same as f everywhere, except for x, where it has the value y. Let $\_$ denote an item, of which the value is irrelevant, and ϵ the empty word.

A *labelled transition system* (LTS) is a tuple $\mathcal{A} = (Q, \Sigma, \rightarrow, q_0)$, where Q is a set of states (a.k.a. configurations), Σ is an alphabet of transition labels, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation and q_0 is the initial state. A path of $\mathcal{A}$ is a finite transition sequence $q_0 \xrightarrow{\beta_1} q_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_k} q_k$ with $k \geq 0$. A trace of $\mathcal{A}$ is a finite sequence $t = \beta_1 \cdot \beta_2 \cdot \ldots \cdot \beta_k$ with $k \geq 0$ if there exists a path $q_0 \xrightarrow{\beta_1} q_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_k} q_k$ of $\mathcal{A}$.

2.2 Concurrent Data Structures and Client Programs

A concurrent data structure provides a number of methods for accessing the data structure. A client program is a program that interacts with concurrent data structures. Concurrent data structures and client programs may contain private memory locations for their own uses. For simplicity of notations, we assume that a method has just one argument and one return value (if it returns).

Given a finite set $\mathcal{X}$ of memory locations, a finite set $\mathcal{M}$ of method names and a finite data domain $\mathcal{D}$, the set *PCom* of primitive commands has the forms below:

$$\begin{aligned} textitPCom ::= \tau \mid \mathit{read}(x, a) \mid \mathit{write}(x, a) \mid \mathit{cas\_suc}(x, a, b) \mid \\ \mathit{cas\_fail}(x, a, b) \mid \mathit{call}(m, a) \mid \mathit{return}(m, a) \end{aligned}$$

where $a, b \in \mathcal{D}, x \in \mathcal{X}$ and $m \in \mathcal{M}$. Herein, τ is the internal command. A *cas* (compare-and-swap) command compresses a read and a write commands into a single one, which is meant to be executed atomically. A successful *cas* command $\mathit{cas\_suc}(x, a,$

$b)$ changes the value of x from a to b, while a failed *cas* command *cas_fail*(x, a, b) does nothing and happens only when the value of x is not a.

A concurrent data structure $\mathcal{L}$ can then be defined as a tuple $\mathcal{L} = (\mathcal{X}_{\mathcal{L}}, \mathcal{M}_{\mathcal{L}}, \mathcal{D}_{\mathcal{L}}, Q_{\mathcal{L}}, \rightarrow_{\mathcal{L}})$, where $\mathcal{X}_{\mathcal{L}}$, $\mathcal{M}_{\mathcal{L}}$ and $\mathcal{D}_{\mathcal{L}}$ are a finite memory location set, a finite method name set and a finite data domain of $\mathcal{L}$ respectively; $Q_{\mathcal{L}} = \bigcup_{m \in \mathcal{M}_{\mathcal{L}}} Q_m$ is the union of disjoint finite sets Q_m of program positions of each method $m \in \mathcal{M}_{\mathcal{L}}$; $\rightarrow_{\mathcal{L}} = \bigcup_{m \in \mathcal{M}_{\mathcal{L}}} \rightarrow_m$ is the union of disjoint transition relations of each method $m \in \mathcal{M}_{\mathcal{L}}$. Let $\mathit{PCom}_{\mathcal{L}}$ be the set of primitive commands (except call and return commands) upon $\mathcal{X}_{\mathcal{L}}$, $\mathcal{M}_{\mathcal{L}}$ and $\mathcal{D}_{\mathcal{L}}$. Then, for each $m \in \mathcal{M}_{\mathcal{L}}$, $\rightarrow_m \subseteq Q_m \times \mathit{PCom}_{\mathcal{L}} \times Q_m$; while for each $a \in \mathcal{D}_{\mathcal{L}}$ there exists an initial state $\mathit{is}_{(m,a)}$ and a final state $\mathit{fs}_{(m,a)}$ in Q_m such that there are neither incoming transitions to $\mathit{is}_{(m,a)}$ nor outgoing transitions from $\mathit{fs}_{(m,a)}$ in $\rightarrow_m$. $\mathit{is}_{(m,a)}$ represents that concurrent data structure begins to execute method m with argument a, and $\mathit{fs}_{(m,a)}$ represents that method m has finished its execution and then a return action with return value a can occur.

A client program $\mathcal{C}$ can then be defined as a tuple $\mathcal{C} = (\mathcal{X}_{\mathcal{C}}, \mathcal{M}_{\mathcal{C}}, \mathcal{D}_{\mathcal{C}}, Q_{\mathcal{C}}, \rightarrow_{\mathcal{C}})$ where $\mathcal{X}_{\mathcal{C}}$, $\mathcal{M}_{\mathcal{C}}$, $\mathcal{D}_{\mathcal{C}}$ and $Q_{\mathcal{C}}$ are a finite memory location set, a finite method name set and a final data domain of $\mathcal{C}$ and a finite program position set, respectively. Let $\mathit{PCom}_{\mathcal{C}}$ be the set of primitive commands upon $\mathcal{X}_{\mathcal{C}}$, $\mathcal{M}_{\mathcal{C}}$ and $\mathcal{D}_{\mathcal{C}}$. Then, $\rightarrow_{\mathcal{C}} \subseteq Q_{\mathcal{C}} \times \mathit{PCom}_{\mathcal{C}} \times Q_{\mathcal{C}}$ is a transition relation of $\mathcal{C}$.

A most general client is a special client program that is designed to exhibit all the possible behaviors of a concurrent data structure. A most general client $\mathcal{MGC}$ can be formally defined as a client $(\{\}, \mathcal{M}_{\mathcal{C}}, \mathcal{D}_{\mathcal{C}}, \{q_c, q'_c\}, \rightarrow_{\mathit{mgc}})$. Here $\rightarrow_{\mathit{mgc}} = \{(q_c, \mathit{call}(m, a), q'_c), (q'_c, \mathit{return}(m, a), q_c, | m \in \mathcal{M}_{\mathcal{C}}, a \in \mathcal{D}_{\mathcal{C}}\}$ is a transition relation. Intuitively, a most general client simply repeatedly calls an arbitrary method with an arbitrary argument for arbitrarily many times.

2.3 TSO Operational Semantics

Let us introduce the operational semantics of concurrent system on TSO memory model, which are used to define TSO-to-TSO linearizability. This operational semantics extends traditional operational semantics of TSO (shown later in this subsection) by introducing flushCall and flushReturn actions, which are used to record time point of when write actions in a method begins to become visible to client, and when this process is complete.

Suppose a concurrent system $C(\mathcal{L})$ that consists of n processes, each of which runs a client program $\mathcal{C}_i = (\mathcal{X}_{\mathcal{C}}, \mathcal{M}, \mathcal{D}, Q_{\mathcal{C}_i}, \rightarrow_{\mathcal{C}_i})$ on a separate processor for $1 \leq i \leq n$, and all the client programs interact with the same concurrent data structure $\mathcal{L} = (\mathcal{X}_{\mathcal{L}}, \mathcal{M}, \mathcal{D}, Q_{\mathcal{L}}, \rightarrow_{\mathcal{L}})$. The operational semantics of the concurrent system $C(\mathcal{L})$ on the TSO memory model is defined as an LTS $[\![C(\mathcal{L}), n]\!]_{\mathit{tt}}$[3] $= (\mathit{Conf}_{\mathit{tt}}, \Sigma_{\mathit{tt}}, \rightarrow_{\mathit{tt}}, \mathit{InitConf}_{\mathit{tt}})$, where $\mathit{Conf}_{\mathit{tt}}, \Sigma_{\mathit{tt}}, \rightarrow_{\mathit{tt}}, \mathit{InitConf}_{\mathit{tt}}$ are defined as follows.

Each configuration of $\mathit{Conf}_{\mathit{tt}}$ is a tuple (p, d, u), where

- $p : \{1, \ldots, n\} \rightarrow Q_{\mathcal{C}i} \cup (Q_{\mathcal{L}} \times Q_{\mathcal{C}i})$ represents control states of each process. $p(i) = q_c \in Q_{\mathcal{C}i}$ represents that process i is executing client position q_c, while

[3] tt represents TSO-to-TSO linearizability.

$p(i) = (q_l, q_c)$ represents that process i is executing position q_l of concurrent data structure, and q_c is the program position of the client program of process i.
- $d : \mathcal{X}_\mathcal{L} \cup \mathcal{X}_\mathcal{C} \rightarrow \mathcal{D}$ is the valuation of memory locations of concurrent data structure and client programs;
- $u : \{1, \ldots, n\} \rightarrow (\{(x, a) | x \in \mathcal{X}_\mathcal{L} \cup \mathcal{X}_\mathcal{C} \wedge a \in \mathcal{D}\} \cup \{call, ret\})^*$ represents contents of each processor-local store buffer; each processor-local store buffer may contain a finite sequence of pending write, pending call or pending return actions.

Σ_{tt} is a set of actions in the following forms:

$$\Sigma_{tt} ::= \tau(i) \mid read(i, x, a) \mid write(i, x, a) \mid cas(i, x, a, b) \mid \\ flush(i, x, a) \mid call(i, m, a) \mid return(i, m, a) \mid \\ flushCall(i) \mid flushReturn(i)$$

where $1 \leq i \leq n, m \in \mathcal{M}$, $x \in \mathcal{X}_\mathcal{L} \cup \mathcal{X}_\mathcal{C}$ and $a, b \in \mathcal{D}$.

The relation T is used to define the transitions occur from concurrent data structures or client programs and is defined as $T = \{((q_{l1}, q_c), \alpha, (q_{l2}, q_c)) | q_{l1}, q_{l2} \in Q_\mathcal{L}, q_c \in Q_{C_i}$ for some $1 \leq i \leq n, q_{l1} \xrightarrow{\alpha}_\mathcal{L} q_{l2}\} \cup \{(q_{c1}, \alpha, q_{c2}) | q_{c1}, q_{c2} \in Q_{C_i}$ for some $1 \leq i \leq n, q_{c1} \xrightarrow{\alpha}_{\mathcal{C}i} q_{c2}$, and α is not a call or return action $\}$. The transition relation $\rightarrow_{tt}$ is the least relation satisfying the transition rules shown in Fig. 1 for each $1 \leq i \leq n$.

- *Tau* rule: A τ transition only influences control state of one process.
- *Read* rule: A function $lookup(u, d, i, x)$ is used to search for the latest value of x from its processor-local store buffer or the main memory, i.e.,

$$lookup(u,d,i,x) = \begin{cases} a & \text{if } u(i) \uparrow_{\Sigma_x} = (x, a) \cdot l, \textit{ for some } l \in \Sigma_x^* \\ d(x) & \text{otherwise} \end{cases}$$

  where $\Sigma_x = \{(x, a) | x \in \mathcal{X}_\mathcal{L} \cup \mathcal{X}_\mathcal{C}, a \in \mathcal{D}\}$ is the set of pending write actions for x. Read action will takes the latest value of x from processor-local store buffer if possible, otherwise, it looks up the value in memory.
- *Write* rule: A write action will insert a pair of location and value to the tail of its processor-local store buffer.
- *Cas-Suc* and *Cas-Fail* rules: A *cas* command can only be executed when the processor-local store buffer is empty and thus forces current process to clear its store buffer in advance. A successful *cas* command will change the value of memory location x immediately while a failed *cas* command does not change memory.
- *Flush* rule: The memory system may decide to flush the entry at the head of processor-local store buffer to memory at any time.
- *Call* and *Return* rules: To deal with *call* command, a call marker is added into the tail of processor-local store buffer and current process starts to execute the initial position of method m and parameter a. When the process comes to the final position of method m it can launch a *return* action, add a return marker to the tail of processor-local store buffer and start to execute the client program.
- *Flush-Call* and *Flush-Return* rules: A call or return marker can be discarded when it is at the head of a processor-local store buffer. Such actions are used to define TSO-to-TSO linearizability only.

$$\frac{T(p(i), c, q'_i,), c = \tau}{(p, d, u) \xrightarrow{\tau(i)}_{tt} (p[i : q'_i], d, u)} \textit{Tau}$$

$$\frac{T(p(i), c, q'_i,), c = \textit{read}(x, a), \textit{lookup}(u, d, i, x) = a}{(p, d, u) \xrightarrow{\textit{read}(i,x,a)}_{tt} (p[i : q'_i], d, u])} \textit{Read}$$

$$\frac{T(p(i), c, q'_i,), c = \textit{write}(x, a), u(i) = l}{(p, d, u) \xrightarrow{\textit{write}(i,x,a)}_{tt} (p[i : q'_i], d, u[i : (x, a) \cdot l])} \textit{Write}$$

$$\frac{T(p(i), c, q'_i,), c = \textit{cas\_suc}(x, a, b), d(x) = a, u(i) = \epsilon}{(p, d, u) \xrightarrow{\textit{cas}(i,x,a,b)}_{tt} (p[i : q'_i], d[x : b], u)} \textit{Cas-Suc}$$

$$\frac{T(p(i), c, q'_i,), c = \textit{cas\_fail}(x, a, b), d(x) \neq a, u(i) = \epsilon}{(p, d, u) \xrightarrow{\textit{cas}(i,x,a,b)}_{tt} (p[i : q'_i], d, u)} \textit{Cas-Fail}$$

$$\frac{u(i) = l \cdot (x, a)}{(p, d, u) \xrightarrow{\textit{flush}(i,x,a)}_{tt} (p, d[x : a], u[i : l])} \textit{Flush}$$

$$\frac{p(i) = q_{c1}, q_{c1} \xrightarrow{\textit{call}(m,a)}_{\mathcal{C}i} q_{c2}, u(i) = l}{(p, d, u) \xrightarrow{\textit{call}(i,m,a)}_{tt} (p[i : (\textit{is}_{(m,a)}, q_{c2})], d, u[i : \textit{call} \cdot l])} \textit{Call}$$

$$\frac{p(i) = (\textit{fs}_{(m,a)}, q_{cl}), q_{c1} \xrightarrow{\textit{return}(m,a)}_{\mathcal{C}i} q_{c2}, u(i) = l}{(p, d, u) \xrightarrow{\textit{return}(i,m,a)}_{tt} (p[i : q_{c2}], d, u[i : \textit{ret} \cdot l])} \textit{Return}$$

$$\frac{u(i) = l \cdot \textit{call}}{(p, d, u) \xrightarrow{\textit{flushCall}(i)}_{tt} (p, d, u[i : l])} \textit{FlushCall}$$

$$\frac{u(i) = l \cdot \textit{ret}}{(p, d, u) \xrightarrow{\textit{flushReturn}(i)}_{tt} (p, d, u[i : l])} \textit{FlushReturn}$$

Fig. 1. Transition Relation $\rightarrow_{tt}$

The initial configuration $\textit{InitConf}_{tt} \in \textit{Conf}_{tt}$ is a tuple $(p_{\textit{init}}, d_{\textit{init}}, \epsilon^n)$. Here $p_{\textit{init}}$ map each process id to a specific state, $d_{\textit{init}}$ is a valuation for memory locations in $\mathcal{X}_{\mathcal{L}} \cup \mathcal{X}_{\mathcal{C}}$, and ϵ^n initializes each process with an empty buffer. If each client program C_i is a most general client, $[\![C(\mathcal{L}), n]\!]_{tt}$ can be abbreviated as $[\![\mathcal{L}, n]\!]_{tt}$.

3 Correctness Conditions

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

TSO-to-TSO linearizability [7] is a variant of linearizability on the TSO memory model, which relates a concurrent data structure running on the TSO memory model to its abstract implementation running also on the TSO memory model. The notion of history is not enough to describe all interactions between concurrent data structures and client programs, since one of them can exhibit a side effect on the other via a store buffer. Therefore, it introduces flushCall and flushReturn actions, which are used to record time point of when write actions in a method begins to become visible to client, and when this process is complete. Let Σ_{fcal} and Σ_{fret} represent the sets of flushCall and flushReturn actions, respectively. A finite sequence $eh \in (\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})^*$ is an extended history of an LTS $\mathcal{A}$, if there exists a trace t of $\mathcal{A}$ such that $t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})} = eh$. Let *ehistory*$(\mathcal{A})$ denote all the extended histories of $\mathcal{A}$, and $eh|_i$ the projection of *eh* to the actions of the i-th process. Two extended history eh_1 and eh_2 are equivalent, if for each $1 \leq i \leq n$, $eh_1|_i = eh_2|_i$.

Given two extended histories eh_1 and eh_2, eh_1 is *TSO-to-TSO linearizable* to eh_2, if

- eh_1 and eh_2 are equivalent;
- there is a bijection $\pi : \{1, \ldots, |eh_1|\} \rightarrow \{1, \ldots, |eh_2|\}$ such that for any $1 \leq i \leq |eh_1|$, $eh_1(i) = eh_2(\pi(i))$;
- for any $1 \leq i < j \leq |eh_1|$, if $(eh_1(i) \in \Sigma_{ret} \cup \Sigma_{fret}) \wedge (eh_1(j) \in \Sigma_{cal} \cup \Sigma_{fcal})$, then $\pi(i) < \pi(j)$.

Given two concurrent data structures $\mathcal{L}_1$ and $\mathcal{L}_2$, we say that $\mathcal{L}_2$ *TSO-to-TSO linearizes* $\mathcal{L}_1$, if for any $eh_1 \in$ *ehistory*$([\![\mathcal{L}_1, n]\!]_{tt})$, there exists $eh_2 \in$ *ehistory*$([\![\mathcal{L}_2, n]\!]_{tt})$, such that eh_1 is *TSO-to-TSO linearizable* to eh_2.

k-bounded TSO-to-TSO linearizability is a bounded variant of TSO-to-TSO linearizability, where only k-extended histories are considered. Its definition is as follows:

Definition 1 **(*k-bounded TSO-to-TSO linearizability*).** *Given concurrent data structures $\mathcal{L}$ and $\mathcal{L}'$, $\mathcal{L}'$ k-bounded TSO-to-TSOlinearizes $\mathcal{L}$ for n processes, if for each k-extended history eh $\in$ k-ehistory$([\![\mathcal{L}, n]\!]_{tt})$, there exists a k-extended history eh$'$ $\in$ ehistory$([\![\mathcal{L}', n]\!]_{tt})$, such that eh is TSO-to-TSO linearizable to eh$'$.*

The following lemma states that, if if both *k-ehistory*$([\![\mathcal{L}, n]\!]_{tt})$ and *k-ehistory*$([\![\mathcal{L}', n]\!]_{tt})$ are finite, then k-bounded TSO-to-TSO linearizability is decidable. This lemma is obvious from Definition 1.

Lemma 1. *Given k-ehistory$([\![\mathcal{L}, n]\!]_{tt})$ and k-ehistory$([\![\mathcal{L}', n]\!]_{tt})$. It is decidable whether $\mathcal{L}'$ k-bounded TSO-to-TSO linearizes $\mathcal{L}$ for n processes.*

Proof. (Sketch)

Given a k-extended history *eh* $\in$ *k-ehistory*$([\![\mathcal{L}, n]\!]_{tt})$ and a k-extended history *eh*$'$ $\in$ *k-ehistory*$([\![\mathcal{L}', n]\!]_{tt})$, we can decide whether *eh* is TSO-to-TSO linearizable to *eh*$'$ as follows:

- If they are not equivalent, then obviously *eh* is not TSO-to-TSO linearizable to *eh*$'$.
- Otherwise, we can enumerate all bijection between $\{1, \ldots, |eh|\}$ and $\{1, \ldots, |eh'|\}$, and check whether this bijection fit the requirement in definition of TSO-to-TSO linearizability. If there is no bijection fits, then *eh* is not TSO-to-TSO linearizable to *eh*$'$. Otherwise, *eh* is TSO-to-TSO linearizable to *eh*$'$.

Since *k-ehistory*$(\llbracket \mathcal{L}, n \rrbracket_{tt})$ and *k-ehistory*$(\llbracket \mathcal{L}', n \rrbracket_{tt})$ are both finite sets, and the number of their elements is bounded by $\mathcal{L}$, n, $\mathcal{D}$ and k, we can enumerate all such pairs (eh, eh') of extended histories above and decide them for k-bounded TSO-to-TSO linearizability. If there exists $eh \in$ *k-ehistory*$(\llbracket \mathcal{L}, n \rrbracket_{tt})$, such that for each $eh' \in$ *k-ehistory*$(\llbracket \mathcal{L}', n \rrbracket_{tt})$, we decide that *eh* is not TSO-to-TSO linearizable to eh', then $\mathcal{L}'$ is not k-bounded TSO-to-TSO linearizes $\mathcal{L}$ for n processes. Otherwise, $\mathcal{L}'$ k-bounded TSO-to-TSO linearizes $\mathcal{L}$ for n processes. □

4 A Modified Operational Semantics For FlushCall and FLushReturn Actions

A call action in $\llbracket \mathcal{L}, n \rrbracket_{tt}$ has two roles that must be taken simultaneously. The first role is to change the control state of the calling process, and the second role is to insert a call marker into buffer. The case of return actions is similar. It seems hard to investigate decidability of k-bounded TSO-to-TSO linearizability with such "composed" call and return actions.Therefore, we separate a "composed" call action into a "pure" call action that only change the control state of the calling process, and a immediately followed write action that inserts a call marker into buffer. When the call marker is flushed by the TSO mechanism, we also launch a flushCall action. In this way, we mimic a "composed" call action with a "pure" call action and a write action.

According to this idea, we propose a modified operational semantics $\llbracket C_f(\mathit{Mod}(\mathcal{L})), n \rrbracket_f$. Let z_f be a new memory location. $\llbracket C_f(\mathit{Mod}(\mathcal{L})), n \rrbracket_f$ mimics a "composed" call action with a "pure" call action and a write action to z_f, and when item of z_f is flushed, it launches a flushCall action. The case for return action is similar. Assume that $\mathcal{L} = (\mathcal{X}_{\mathcal{L}}, \mathcal{M}, \mathcal{D}_{\mathcal{L}}, Q_{\mathcal{L}}, \rightarrow_{\mathcal{L}})$, then, C_f and $\mathit{Mod}(\mathcal{L})$ are defined as follows:

- The function C_f maps each $1 \leq i \leq n$ to a modified most general client program $\mathit{ModMGC} = (\{z_f\}, \mathcal{M}, \mathcal{D}_{\mathcal{L}} \cup \{\mathit{call}, \mathit{ret}\}, \{q_c, q_c', q_c''\}, \rightarrow)$. The transition relation $\rightarrow$ is defined as follows: $\rightarrow = \{(q_c, \mathit{call}(m,a), q_c'),\ (q_c', \mathit{return}(m,a), q_c''),\ (q_c'', \mathit{write}(z_f, \mathit{ret}), q_c) |\ m \in \mathcal{M}, a \in \mathcal{D}_{\mathcal{L}}\}$. Here we add a $\mathit{write}(z_f, \mathit{ret})$ transition to the most general client after a return transition, and this $\mathit{write}(z_f, \mathit{ret})$ transition is used to simulate the insertion of return marker in $\llbracket \mathcal{L}, n \rrbracket_{tt}$.
- The concurrent data structure $\mathit{Mod}(\mathcal{L}) = (\mathcal{X}_{\mathcal{L}} \cup \{z_f\}, \mathcal{M}, \mathcal{D}_{\mathcal{L}} \cup \{\mathit{call}, \mathit{ret}\}, Q_{\mathcal{L}} \cup \{q^{\mathcal{L}}\}, \rightarrow_{\mathcal{L}}')$ slightly change the concurrent data structure $\mathcal{L}$. $\rightarrow_{\mathcal{L}}'$ is generated from $\rightarrow_{\mathcal{L}}$ as follows: For each $a \in \mathcal{D}_{\mathcal{L}}$, $m \in \mathcal{M}$ and state $q \in Q_{\mathcal{L}}$, if $\mathit{is}_{(m,a)} \xrightarrow{\mathit{act}}_{\mathcal{L}} q$, then $\rightarrow_{\mathcal{L}}'$ replaces this transition with the transitions $\mathit{is}_{(m,a)} \xrightarrow{\mathit{write}(z_f, \mathit{call})}{}'_{\mathcal{L}} q^{\mathcal{L}}$ and $q^{\mathcal{L}} \xrightarrow{\mathit{act}}{}'_{\mathcal{L}} q$. Here we add $\mathit{write}(z_f, \mathit{call})$ transitions as the first transition of a method, which are used to simulate the insertion of call marker in $\llbracket \mathcal{L}, n \rrbracket_{tt}$.

The modified operational semantics $\llbracket C_f(\mathit{Mod}(\mathcal{L})), n \rrbracket_f$[4] $= (\mathit{Conf}_f, \Sigma_f, \rightarrow_f, \mathit{InitConf}_f)$. Here Conf_f is generated from Conf_{tt} by introducing memory location z_f and control state q_c'' of client program, Σ_f are generated from Σ_{tt} by introducing operation to z_f,

[4] f represents that this operational semantics is proposed for flushCall and flushReturn actions.

InitConf$_f$ is generated from *InitConf*$_{tt}$ by introducing valuation to z_f. The transition relation $\rightarrow_f$ is generated from $\rightarrow_t$ (Recall that $\rightarrow_t$ is generated from $\rightarrow_{tt}$ in Section 3) by discarding the *Flush* rule and adding the following rules:

$$\frac{u(i) = l \cdot (x, a), x \neq z_f}{(p, d, u) \xrightarrow{\mathit{flush}(i,x,a)}_f (p, d[x : a], u[i : l])} \mathit{Flush'}$$

$$\frac{u(i) = l \cdot (z_f, \mathit{call})}{(p, d, u) \xrightarrow{\mathit{flushCall}(i)}_f (p, d, u[i : l])} \mathit{FlushCall'}$$

$$\frac{u(i) = l \cdot (z_f, \mathit{ret})}{(p, d, u) \xrightarrow{\mathit{flushReturn}(i)}_f (p, d, u[i : l])} \mathit{FlushReturn'}$$

The following lemma states that the extended histories of $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f$ equals the extended histories of $[\![\mathcal{L}, n]\!]_{tt}$. Thus, the modified operational semantics $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f$ can be used to investigate the decidability of TSO-to-TSO linearizability.

Lemma 2. *$\mathit{ehistory}([\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f) = \mathit{ehistory}([\![\mathcal{L}, n]\!]_{tt})$.*

Proof. (Sketch)

Let $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]'_f$ be an LTS that is generated from $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f$ by transforming all write actions of z_f into τ action. We prove this lemma by constructing a weak bisimulation relation between states of $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]'_f$ and $[\![\mathcal{L}, n]\!]_{tt}$.

A relation $\approx$ is defined as follows: given configuration (p_f, d_f, u_f) of $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]'_f$ and configuration (p_{tt}, d_{tt}, u_{tt}) of $[\![\mathcal{L}, n]\!]_{tt}$, $(p_f, d_f, u_f) \approx (p_e, d_e, u_e)$, if

- For p_f and p_{tt}, for each $1 \leq i \leq n$, one of the following case holds:
  - $p_f(i) = q_c \vee p_f(i) = q''_c$ and $p_{tt}(i) = q_c$.
  - $p_f(i) = (is_{(m,a)}, q'_c) \vee p_f(i) = (q^{\mathcal{L}}, q'_c)$ for $m \in \mathcal{M}$ and $a \in \mathcal{D}_{\mathcal{L}}$ and $p_{tt}(i) = (is_{(m,a)}, q'_c)$.
  - If $p_f(i) \notin \{q_c, q''_c, (q^{\mathcal{L}}, q'_c), (is_{(m,a)}, q'_c) | m \in \mathcal{M}, a \in \mathcal{D}_{\mathcal{L}}\}$ and $p_{tt}(i) \notin \{q_c, (is_{(m,a)}, q'_c) | m \in \mathcal{M}, a \in \mathcal{D}_{\mathcal{L}}\}$, then $p_f(i) = p_e(i)$.
- For d_f and d_{tt}, for each $x \in \mathcal{X}_{\mathcal{L}}$, $d_f(x) = d_{tt}(x)$.
- Let $\mathit{ch}(s)$ be a sequence that is generated from s by transforming every (z_f, call) into *call*, and transforming every (z_f, ret) into *ret*. For u_f and u_{tt}, for each $1 \leq i \leq n$, one of the following case holds:
  - If $p_f(i) = q''_c$ and $p_{tt}(i) = q_c$, then $\mathit{ret} \cdot \mathit{ch}(u_f(i)) = u_{tt}(i)$.
  - If $p_f(i) = (is_{(m,a)}, q'_c)$ and $p_{tt}(i) = (is_{(m,a)}, q'_c)$, then $\mathit{call} \cdot \mathit{ch}(u_f(i)) = u_{tt}(i)$.
  - Otherwise, $\mathit{ch}(u_f(i)) = u_{tt}(i)$.

It is not hard to see that $\approx$ is a weak bisimulation relation by consider all kinds of transitions. Therefore, it is obvious that $\mathit{ehistory}([\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f) = \mathit{ehistory}([\![\mathcal{L}, n]\!]_{tt})$. $\square$

5 Perfect/Lossy Channel Machines

A classical channel machine is a finite control machine equipped with channels of unbounded sizes. It can perform send and receive operations on its channels. A lossy channel machine is a channel machine where arbitrary many items in its channels may be lost nondeterministically at any time without any notification. In this section we sketch our definition of (S, K)-channel machines, which slightly differs from the definition of channel machines in [14].

The channel machines defined in [14] extend classical channel machines in the following aspects:

- Each transition is guarded by a condition about whether the content of a channel is in a regular language.
- A substitution to the content of a channel may be performed before a send operation on the channel.
- A set of specific symbols, called “strong symbols”, are introduced that are not allowed to be lost, but the number of strong symbols in a channel is always bounded.

In this paper, we extend the channel machines defined in [14] with multiple sets of strong symbols, while the number of strong symbols in a channel from the same strong symbol set is separately bounded.

Let $\mathcal{CH}$ be the finite set of channel names and $\Sigma_{\mathcal{CH}}$ be a finite alphabet of channel contents. The content of a channel is a finite sequence over $\Sigma_{\mathcal{CH}}$. For a given channel $c \in \mathcal{CH}$, a regular guard on channel c is a constraint of the form $c \in L$, where $L \subseteq \Sigma_{\mathcal{CH}}^*$ is a regular set of sequences. For a sequence $u \in \Sigma_{\mathcal{CH}}^*$ we write $u \models c \in L$ if $l \in L$. For notational convenience, we write $a \in c$ instead of $c \in \Sigma_{\mathcal{CH}}^* \cdot a \cdot \Sigma_{\mathcal{CH}}^*$, $c = \epsilon$ instead of $c \in \{\epsilon\}$ and $c : \Sigma'$ instead of $c \in \Sigma'^*$ for any subset Σ' of $\Sigma_{\mathcal{CH}}$. A regular guard over $\mathcal{CH}$ associates a regular guard for each channel of $\mathcal{CH}$. Let *Guard*$(\mathcal{CH})$ be the set of regular guards over $\mathcal{CH}$. The definition of $\models$ can be extended as follows: for $g \in$ *Guard*$(\mathcal{CH})$ and $u \in \mathcal{CH} \to \Sigma_{\mathcal{CH}}^*$, we write $u \models g$, if $u(c) \models g(c)$ for each $c \in \mathcal{CH}$.

Given a channel $c \in \mathcal{CH}$, a channel operation on c is either a *nop* (no operation), or an $c?a$ operation for some $a \in \Sigma_{\mathcal{CH}}$ (receive operation), or an $c[\sigma]!a$ operation (send operation) where σ is a substitution over $\Sigma_{\mathcal{CH}}$ and a is a element of $\Sigma_{\mathcal{CH}}$. We write $c!a$ instead of $c[\sigma]!a$ when σ is the identity substitution. For every $u, u' \in \Sigma_{\mathcal{CH}}^*$, we have $[\![nop]\!](u, u')$ if $u = u'$, $[\![c[\sigma]!a]\!](u, u')$ if $u' = a \cdot u[\sigma]$, $[\![c?a]\!](u, u')$ if $u = u' \cdot a$. A channel operation over $\mathcal{CH}$ is a mapping that associates with each channel c a channel operation on c. Let *Op*$(\mathcal{CH})$ be the set of channel operations over $\mathcal{CH}$. The definition of $[\![op]\!]$ can be extended as follows: for $op \in$ *Op*$(\mathcal{CH})$ and $u, u' \in \mathcal{CH} \to \Sigma_{\mathcal{CH}}^*$, we have $[\![op]\!](u, u')$, if $[\![op(c)]\!](u(c), u'(c))$ holds for each $c \in \mathcal{CH}$.

A *channel machine* is formally defined as a tuple $M = (Q, \mathcal{CH}, \Sigma_{\mathcal{CH}}, \Lambda, \Delta)$, where (1) Q is a finite set of states, (2) $\mathcal{CH}$ is a finite set of channel names, (3) $\Sigma_{\mathcal{CH}}$ is an alphabet for channel contents, (4) Λ is a finite set of transition labels, and (5) $\Delta \subseteq Q \times (\Lambda \cup \{\epsilon\}) \times$ *Guard*$(\mathcal{CH}) \times$ *Op*$(\mathcal{CH}) \times Q$ is a finite set of transitions.

We say a sequence $l_1 = a_1 \cdot \ldots \cdot a_u$ is a subword of another sequence $l_2 = b_1 \cdot \ldots \cdot b_v$, if there exists $i_1 < \ldots < i_u$, such that $a_j = b_{i_j}$ for each j. Let $S = \langle s_1, \ldots, s_m \rangle$ be a vector of sets with $s_i \subseteq \Sigma_{\mathcal{CH}}$ for $1 \le i \le m$, and $K = \langle k_1, \ldots, k_m \rangle$ be a vector of nature numbers or ∞. S is the sets of strong symbols that must be kept in transition, and K is the bounds for each set of strong symbols in S. For sequences $u, v \in \Sigma^*_{\mathcal{CH}}$, $u \preceq^K_S v$ holds if (1) u is a subword of v, (2) for each i, $u \uparrow_{s_i} = v \uparrow_{s_i}$ and (3) for each j, $|u \uparrow s_j| \le k_j$. This relation can be extended as follows: For every $u, v \in \mathcal{CH} \to \Sigma^*_{\mathcal{CH}}$, $u \preceq^K_S v$ holds, if $u(c) \preceq^K_S v(c)$ holds for each $c \in \mathcal{CH}$.

A *(S,K)-channel machine* (abbreviated as *(S,K)-CM*) is a channel machine $M = (Q, \mathcal{CH}, \Sigma_{\mathcal{CH}}, \Lambda, \Delta)$ with the strong symbol restriction (S, K). Its semantics is defined as an *LTS* $(\mathit{Conf}_M, \Lambda, \to_M, \mathit{initConf}_M)$. A configuration of Conf_M is a pair (q, u) where $q \in Q$, $u : \mathcal{CH} \to \Sigma^*_{\mathcal{CH}}$, and it satisfies the strong symbol restriction(S, K), i.e., for each c and i, $|u(c) \uparrow s_i| \le k_i$. The transition relation $\to_M$ is defined as follows: given $q, q' \in Q$ and $u, u' \in \mathcal{CH} \to \Sigma^*_{\mathcal{CH}}$, $(q, u) \xrightarrow{\alpha}_M (q', u')$, if there exists g and *op*, such that $(q, \alpha, g, \mathit{op}, q') \in \Delta$, $u \models g$ and $[\![\mathit{op}]\!](u, u')$. Similarly, a *(S,K)-lossy channel machine* (abbreviated as *(S,K)-LCM*) is a channel machine M with lossy channels and the strong symbol restriction (S, K). Its semantics is defined as an *LTS* $(\mathit{Conf}_M, \Lambda, \to_{\mathit{(M,S,K)}}, \mathit{initConf}_M)$. The transition relation $\to_{\mathit{(M,S,K)}}$ is defined as follows: $(q, u) \xrightarrow{\alpha}_{\mathit{(M,S,K)}} (q', u')$, if there exists $v, v' \in \mathcal{CH} \to \Sigma^*_{\mathcal{CH}}$, such that $v \preceq^K_S u$, $(q, v) \xrightarrow{\alpha}_M (q', v')$ and $u' \preceq^K_S v'$. Let $\to^*_M$ and $\to^*_{\mathit{(M,S,K)}}$ be the transition closure of $\to_M$ and $\to_{\mathit{(M,S,K)}}$.

Given a channel machine M, we say that $(q_0, u_0) \cdot \alpha_1 \cdot (q_1, u_1) \cdot \ldots \cdot \alpha_w \cdot (q_w, u_w)$ is a finite run of M from (q, u) to (q', u'), if (1) $(q_0, u_0) = (q, u)$, (2) $(q_i, u_i) \xrightarrow{\alpha_{i+1}}_M (q_{i+1}, u_{i+1})$ for each i and (3) $(q_w, u_w) = (q', u')$. We say that l is a trace of a finite run ρ if $l = \rho \uparrow_\Lambda$. Given $q, q' \in Q$, let $T^{S,K}_{q,q'}(M)$ denote the set of traces of all finite runs of a (S, K)-*CM* M from the configuration $(q, \epsilon^{|n|})$ to the configuration $(q', \epsilon^{|n|})$. For $(S, K) - \mathit{LCM}\ M$, the notations of finite run and its trace are defined as in the non-lossy case by replacing $\to_M$ with $\to_{(M,S,K)}$. Let $\mathit{LT}^{S,K}_{q,q'}(M)$ denote the set of traces of all finite runs of (S, K)-*LCM* M from the configuration $(q, \epsilon^{|n|})$ to the configuration $(q', \epsilon^{|n|})$.

For channel machines $M_1 = (Q_1, \mathcal{CH}_1, \Sigma_{\mathcal{CH}}, \Lambda, \Delta_1)$ and $M_2 = (Q_2, \mathcal{CH}_2, \Sigma_{\mathcal{CH}}, \Lambda, \Delta_2)$ such that $\mathcal{CH}_1 \cap \mathcal{CH}_2 = \emptyset$, the product of M_1 and M_2 is also a channel machine $M_1 \otimes M_2 = (Q_1 \times Q_2, \mathcal{CH}_1 \cup \mathcal{CH}_2, \Sigma_{\mathcal{CH}}, \Lambda, \Delta_{12})$, where Δ_{12} is defined by synchronizing transitions sharing the same label in Λ under the conjunction of their guards, and letting other transitions asynchronous. The following lemma holds as in [14].

Lemma 3. *Given channel machines* $M_1 = (Q_1, \mathcal{CH}_1, \Sigma_{\mathcal{CH}}, \Lambda, \Delta_1)$ *and* $M_2 = (Q_2, \mathcal{CH}_2, \Sigma_{\mathcal{CH}}, \Lambda, \Delta_2)$*, let* $q_1, q_1' \in Q_1$, $q_2, q_2' \in Q_2$, $q = (q_1, q_2)$, $q' = (q_1', q_2')$*, then* $\mathit{LT}^{S,K}_{q,q'}(M_1 \otimes M_2) = \mathit{LT}^{S,K}_{q_1,q_1'}(M_1) \cap \mathit{LT}^{S,K}_{q_2,q_2'}(M_2)$ *and* $T^{S,K}_{q,q'}(M_1 \otimes M_2) = T^{S,K}_{q_1,q_1'}(M_1) \cap T^{S,K}_{q_2,q_2'}(M_2)$.

Given a (S, K)-*CM* (respectively, (S, K)-*LCM*) M and two states $q, q' \in Q$, a control state reachability problem of M is to determine whether $T^{S,K}_{q,q'}(M) \ne \emptyset$ (respectively, $\mathit{LT}^{S,K}_{q,q'}(M) \ne \emptyset$). As in [14], it can be shown that the control state reachability problem is decidable for (S, K)-*LCM*.

6 A Modified Operational Semantics For Call and Return Actions

In Section 4, we have separated a "composed" call action in $[\![\mathcal{L}, n]\!]_{tt}$ into a call action and a write action of z_f. However, call and return actions are to some extend "beyond" TSO memory model and are quite hard to be dealt with in TSO mechanism. To explain this, it should be noted that on TSO, write actions can be seen by other process only when they are flushed. This implies that, when a process want to notice another process by writing something into buffer, then this message can be seen by other process only when it is flushed. As contrast, a call or return action takes effect as soon as it happens. To make it worse, the order of call and return action may be different from the order how processes use buffer. Since the content of buffer may influence future execution, a process can not notice other process of call or return actions by using *cas* actions, which will clear the buffer. Therefore, it is hard to make a concurrent system to know the order of call and return actions without influence its future execution on TSO.

To make a concurrent system be aware of call and return actions, we modify the operational semantics $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f$ as follows:

- We add a new process into the concurrent system, which is used as an "observer" for call and return actions. This observer process nondeterministically guess call and return actions by writing call and return information with *cas* actions to a new memory location z_w.
- We modify the operational semantics, and make specific *cas* actions of the observer process be bound with call and return actions (we can also say that these *cas* actions mark the call and return actions). That is, a $\mathit{cas}(z_f, \mathit{call}(i, m, a))$ action (resp., $\mathit{cas}(z_f, \mathit{return}(i, m, a))$ action) of the observer process is bound with a $\mathit{call}(i, m, a)$ actions (resp., $\mathit{return}(i, m, a)$).

Formally, let $\mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n) = \{\mathit{call(i,m,a)}, \mathit{return(i,m,a)} | 1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}_{\mathcal{L}}\}$ denote the set of values that are used by the specific *cas* actions to mark the call and return actions in $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f$. Let z_w be a new memory location, which will be used by the specific *cas* actions. Assume that $\mathcal{L} = (\mathcal{X}_{\mathcal{L}}, \mathcal{M}, \mathcal{D}_{\mathcal{L}}, Q_{\mathcal{L}}, \rightarrow_{\mathcal{L}})$. Then, a function$\mathit{Clt}_f$ is defined as follows:

- For each $1 \leq i \leq n$, Clt_f maps process P_i into the client program $\mathit{ModMGC} = (\{z_f\}, \mathcal{M}, \mathcal{D}_{\mathcal{L}} \cup \{\mathit{call}, \mathit{ret}\}, \{q_c, q'_c, q''_c\}, \rightarrow)$, which has been defined in Section 4.
- Clt_f maps process P_{n+1} into the client program $C_{\mathit{marked}} = (\{z_w\}, \mathcal{M}, \mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n), \{q_{\mathit{wit}}\}, \rightarrow_{\mathit{wit}})$. Here $\rightarrow_{\mathit{wit}} = \{(q_{\mathit{wit}}, \mathit{cas\_suc}(z_w, \_, a), q_{\mathit{wit}}) |\ a \in \mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n)\}$ is the transition relation of C_{marked}.

Assume that $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f = (\mathit{Conf}_f, \Sigma_f, \rightarrow_f, \mathit{InitConf}_f)$. According to above idea, we propose a operational semantics $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$, which contains $n{+}1$ processes. $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$[5] $= (\mathit{Conf}_b, \Sigma_b, \rightarrow_b, \mathit{InitConf}_b)$. A configuration of Conf_b is a tuple (p, d, u, mak). Here (p, d, u) is a configuration generated from a configuration of Conf_f by introducing memory location z_w and control of the observer process. $\mathit{mak} \in \{\epsilon\} \cup \mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n)$ is used to ensure that each specific *cas* action bind

[5] b represents bind.

a corresponding call or return action. Σ_b are generated from Σ_f by introducing operation to z_w, *InitConf*$_b$ is generated from *InitConf*$_f$ by introducing valuation to z_w. The transition relation $\rightarrow_b$ is generated from $\rightarrow_f$ as follows:

- For all but *cas*, call and return transitions, if $(p, d, u) \xrightarrow{act}_f (p', d', u')$, then we have $(p, d, u, \epsilon) \xrightarrow{act}_b (p', d', u', \epsilon)$.
- We discard the *Cas-Suc* and *Cas-Fail* rules, and add the following rules:

$$\frac{T(p(i), c, q_i',), c = \mathit{cas\_suc}(x, a, b), 1 \leq i \leq n \wedge x \neq z_w, d(x) = a, u(i) = \epsilon}{(p, d, u, \epsilon) \xrightarrow{cas(i,x,a,b)}_b (p[i : q_i'], d[x : b], u, \epsilon)} \textit{Cas-Suc-B}$$

$$\frac{T(p(i), c, q_i',), c = \mathit{cas\_fail}(x, a, b), 1 \leq i \leq n \wedge x \neq z_w, d(x) \neq a, u(i) = \epsilon}{(p, d, u, \epsilon) \xrightarrow{cas(i,x,a,b)}_b (p[i : q_i'], d, u, \epsilon)} \textit{Cas-Fail-B}$$

$$\frac{T(p(n{+}1), c, q_{n+1}',), c = \mathit{cas\_suc}(z_w, a, b), a, b \in \mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n), d(x) = a, u(n{+}1) = \epsilon}{(p, d, u, \epsilon) \xrightarrow{cas(n+1,z_w,a,b)}_b (p[n{+}1 : q_i'], d[x : b], u, b)} \textit{Cas-MARK}$$

  Note that only the *Cas-MARK* can set the *mak* tuple.

- We discard the *Call* and *Return* rules, and add the following rules:

$$\frac{p(i) = q_{c1}, q_{c1} \xrightarrow{call(m,a)}_{\mathcal{C}i} q_{c2}}{(p, d, u, \mathit{call}(i, m, a)) \xrightarrow{call(i,m,a)}_b (p[i : (\mathit{is}_{(m,a)}, q_{c2})], d, u, \epsilon)} \textit{Call-B}$$

$$\frac{p(i) = (\mathit{fs}_{(m,a)}, q_{c1}), q_{c1} \xrightarrow{return(m,a)}_{\mathcal{C}i} q_{c2}}{(p, d, u, \mathit{return}(i, m, a)) \xrightarrow{return(i,m,a)}_b (p[i : q_{c2}], d, u, \epsilon)} \textit{Return-B}$$

  Note that call or return action need the correct *mak* be set, and they will unset the *mak* tuple.

The following lemma states that the extended histories of $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f$ equals the extended histories of $[\![\mathcal{L}, n]\!]_{tt}$. Thus, the modified operational semantics $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f$ can be used to investigate the decidability of TSO-to-TSO linearizability.

Lemma 4. *ehistory*$([\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b) =$ *ehistory*$([\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f)$.

Proof. (Sketch)

It is obvious that *ehistory*$([\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b) \subseteq$ *ehistory*$([\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f)$, since except for $\mathit{cas}(n{+}1, z_w, \_, \_)$ transitions, each transition in $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$ comes from a transition in $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f)$.

For the opposite direction of this lemma, given a path $pa = (p_0, d_0, u_0) \xrightarrow{act_1}_f (p_1, d_1, u_1) \ldots \xrightarrow{act_w}_f (p_w, d_w, u_w)$ of $[\![C_f(\mathit{Mod}(\mathcal{L})), n]\!]_f$, we can in w steps construct a path pa' of

$[\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_b$, such that the extended history of *pa* equals the extended history of pa'.

Let $pa'_0=(p_0,d_0,u_0,\epsilon)$. For each $1\leq i\leq w$, assume that $pa'_{i\text{-}1}$ ends in $(p_{i\text{-}1},d_{i\text{-}1},u_{i\text{-}1},mak_{i\text{-}1})$ and $(p_{i\text{-}1},d_{i\text{-}1},u_{i\text{-}1})\xrightarrow{act_i}_f(p_i,d_i,u_i)$. If act_i is not call or return action, then pa'_i is obtained from $pa'_{i\text{-}1}$ by adding $(p_{i\text{-}1},d_{i\text{-}1},u_{i\text{-}1},mak_{i\text{-}1})\xrightarrow{act_i}_b(p_i,d_i,u_i,\epsilon)$ transition. Otherwise, pa'_i is obtained from $pa'_{i\text{-}1}$ by adding $(p_{i\text{-}1},d_{i\text{-}1},u_{i\text{-}1},mak_{i\text{-}1})\xrightarrow{cas(n+1,z_w,\_,act_i)}_b(p_{i\text{-}1},d_{i\text{-}1},u_{i\text{-}1},act_i)$ and $(p_{i\text{-}1},d_{i\text{-}1},u_{i\text{-}1},act_i)\xrightarrow{act_i}_b(p_i,d_i,u_i,\epsilon)$ transitions. It is easy to see that $pa'=pa_w$ holds as required. This completes the proof of this lemma. □

Given an extended history *eh*, a marked witness of *eh* in $[\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_b$ is a trace t of $[\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_b$, such that

- $t\uparrow_{(\Sigma_{cal}\cup\Sigma_{ret}\cup\Sigma_{fcal}\cup\Sigma_{fret})}=eh$.
- $t=t_1\cdot t_2$ such that t_1 ends with a call, return, flushCall or flushReturn action, t_2 is a sequence of flush actions, and all the items putted by write actions in t have been flushed.

Similar to the proof of Lemma 4, we can see that, there exists a marked witness of *eh*, if and only if $eh\in ehistory([\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_b)$. By Lemma 4 and Lemma 2, we know that there exists a marked witness of *eh*, if and only if $eh\in ehistory([\![\mathcal{L},n]\!]_{tt})$. Therefore, we can check whether a specific extended history is in $ehistory([\![\mathcal{L},n]\!]_{tt})$ by checking whether there is a marked witness for it in $[\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_b$.

7 Channel Machines For Marked Witness

In this section, we use an example to explain how to use channel machine to simulate behavior of each process of $[\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_b$. Then, we propose the detailed definition of these channel machines.

7.1 Simulating $[\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_f$ with Channel Machines

In this subsection, we use an example to intuitively show how to simulate bounded behaviors (that contains at most k call, return, flushCall and flushReturn actions) of each process i ($1\leq i\leq n{+}1$) of $[\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_b$ with a (S,K)-channel machine M_i^k. Then, in later section we show that the bounded behavior of concurrent system can be simulated by "production" of $M_i^1,\dots M_i^{n+1}$.

Each M_i^k ($1\leq i\leq n{+}1$) contains control state of process i and launch actions according to these control state. M_i^k also nondeterministically guesses the write, call or return actions of other processes. Since write actions to z_f of other process are guessed, M_i^k essentially "guess" flushCall and flushReturn of other processes. Moreover, M_i^k ensures that flush or *cas* action to z_w is immediate followed by a corresponding call or return action. That is, for $1\leq i\leq n$, each $flush(n{+}1,z_w,act)$ (note that $cas(n{+}1,z_w,\_,\_)$ are considered as guess write for M_i^k) is immediate followed by a call or return action *act*; for M_i^{n+1}, each $cas(n{+}1,z_w,\_,act)$ is immediately followed by a corresponding call or return action. M_i^k contains only one channel c_i that is used to store the pending written items according to the total store orders (orders of actions that updates memory locations) in $[\![Clt_f(Mod(\mathcal{L})),n{+}1]\!]_b$. To make it safe to become a lossy channel machine, each item sent to c_i contains the current valuation of all the memory locations, i.e., the current snapshot of the memory. M_i^k also monitors whether a specific extended history has already happens. (shown in the following subsection).

We use the example shown in Fig. 2 to illustrate the main idea of our construction method. Fig. 2 (a) presents a trace of 8-extended history of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), 3]\!]_b$, while Fig. 2 (b),(c),(d) present the corresponding traces of M_1^k, M_2^k, M_3^k, respectively. Each pair of a call and its accompanying return action is associated with a (dashed) line interval that start and end with vertical lines. To emphasize the occurrence of flushCall and flushReturn actions, we associate each pair of a flushCall and its accompanying flushReturn action with a (dashed) line interval that start and end with circles. Here a interval of dashed line means that actions of this interval are generated by guessing in M_i^k, while a interval of ordinary line means that actions of this interval are generated according to process i in M_i^k.

Let us begin to explain Fig. 2. $w(x)1$ is a action that write 1 to x; $r(x)1$ is an action that reads 1 from x; $f(x)1$ is a flush action that changes the value of x to 1; $ca(y)1$ is a *cas* action that changes the value of y to 1. For simplicity, we use $a_1, \dots, a_4$ to represent four call or return information written to z_w, and use $b_1, \dots, b_4$ to represent four call or return markers written to z_f. $b_1 = b_2 = \mathit{call}$, and $b_3 = b_4 = \mathit{ret}$. Let method M_1 be called with argument arg_1 and return value rv_1 in process 1, and let method M_2 be called with argument arg_2 and return value rv_2 in process 2. In detail, $a_1 = \mathit{call}(1, M_1, \mathit{arg}_1)$, $a_2 = \mathit{call}(2, M_2, \mathit{arg}_2)$, $a_3 = \mathit{return}(1, M_1, \mathit{rv}_1)$ and $a_4 = \mathit{return}(2, M_2, \mathit{rv}_2)$. We write flushCall or flushReturn as $f(z_f)b_{\_}$ to emphasize that they are generated by flush z_f items. Also note that the actions in Fig. 2 (a) use individual values, while the the actions in Fig. 2 (b),(c),(d) use snapshots of the memory. In Fig. 2 (b),(c),(d) we use 1, a_i and b_i only for simplicity, which are essentially corresponding snapshot of the memory.

The requirements for t_i $(1 \leq i \leq 3)$ are as follows:

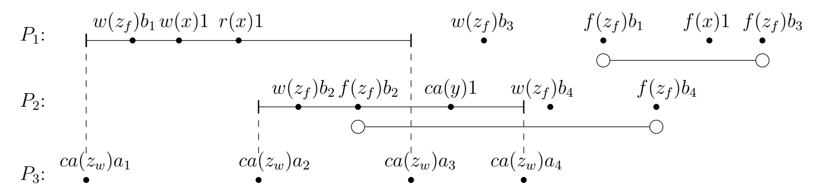
- t_i and t has the same extended history and same total store order (we do not distinguish *flush*(i, x, a) and *cas*$(i, x, \_, a)$).
- The flush or *cas* action of z_w in t_i must be immediately followed by a corresponding call or return actions.
- The actions of process i in t_i equals the actions of process i in t. When t_i launch an action of process i, the valuation of memory locations must equals to the valuation of memory locations of the same action in t.

The total store order in t are $ca(z_w)a_1, ca(z_w)a_2, f(z_f)b_2, ca(z_w)a_3, ca(y)1, ca(z_w)a_3, f(z_f)b_1, f(z_f)b_4, f(x)1, f(z_f)b_3$.

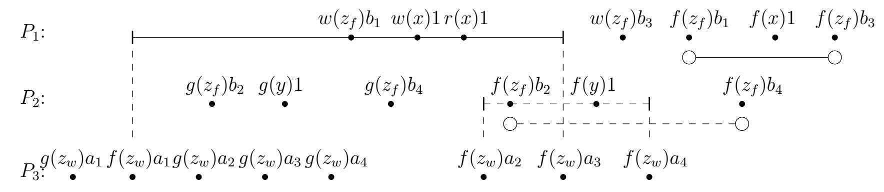
7.2 Construction of M_i^k $(1 \leq i \leq n)$

In this subsection, we present the formal definition of the channel machine M_1^k $(1 \leq i \leq n)$, which simulates the behavior of process i of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), \mathit{n+1}]\!]_b$ and monitors whether a specific k-extended history happens.

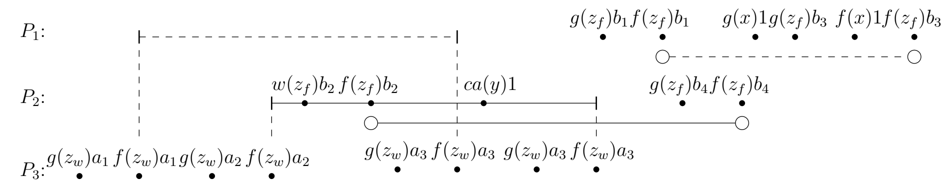
Given a k-extended history $\mathit{eh} = \mathit{act}_1 \cdot \ldots \cdot \mathit{act}_l$, where method is chosen from $\mathcal{M}$, data domain is $\mathcal{D}$ and process id of each act_i is in $\{1, \dots, n\}$, we construct a



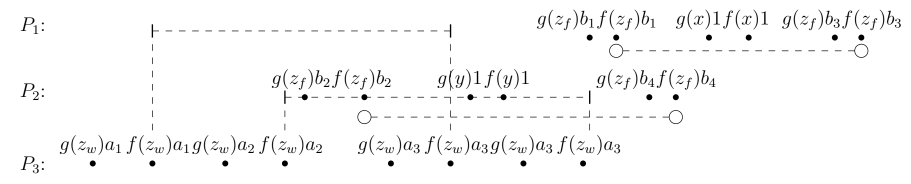
(a) a trace t of $[\![Clt_f(Mod(\mathcal{L})), 3]\!]_f$



(b) trace t_1 of M_1^k for t



(c) trace t_2 of M_2^k for t



(d) trace t_3 of M_3^k for t

Fig. 2. traces of M_1^k, M_2^k and M_3^k for a trace t of $[\![Clt_f(Mod(\mathcal{L})), 3]\!]_f$

deterministic finite state automaton $\mathcal{A}_{eh}$ that accepts *eh*. $\mathcal{A}_{eh} = (Q_s, \Sigma_s, \rightarrow_s, F_s, q_{is})$ is constructed as follows: $Q_s = \{q_{is}, q_s^1, \ldots, q_s^{l\text{-}1}, q_{ends}\}$ is a finite set of states, $\Sigma_s = \{call(i,m,a), return(i,m,a), flushCall(i,m,a), flushReturn(i,m,a) | 1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}\}$ is a set of transition labels, q_{is} is the initial state and $F_s = \{q_{ends}\}$ is the set of accepting states. The transition relation $\rightarrow_s$ contains the following transitions: $q_{is} \xrightarrow{act_1}_s q_s^1$, $q_s^i \xrightarrow{act_{i+1}}_s q_s^{i+1}$ ($1 \leq i \leq l\text{-}2$), and $q_s^{l\text{-}1} \xrightarrow{act_l}_s q_{ends}$.

We now present the formal definition of the channel machine M_i^k. Assume that $\mathcal{L} = (\mathcal{X}_\mathcal{L}, \mathcal{M}, \mathcal{D}_\mathcal{L}, Q_\mathcal{L}, \rightarrow_\mathcal{L})$. Assume that the transition relation of $Mod(\mathcal{L})$ is $\rightarrow'_\mathcal{L}$. Assume that the set of states in *ModMGC* is $\{q_c, q'_c, q''_c\}$. Let *Val* be the set of valuation functions that maps a memory location in $\mathcal{X}_\mathcal{L}$ to a value in $\mathcal{D}_\mathcal{L}$, maps z_w to a value in $markedVal(\mathcal{M}, \mathcal{D}_\mathcal{L}, n)$ and maps z_f to $\{call, ret\}$. Channel machine M_i^k ($1 \leq i \leq n$) is a tuple $(Q_i^k, \{c_i\}, \Sigma, \Lambda, \Delta_i^k)$, where c_i is name of the single channel of M_i^k. Q_i^k, Σ, Λ and Δ_i^k are defined as follows:

Construction of Q_i^k: $Q_i^k = (Q_c \cup (Q_\mathcal{L} \times Q_c)) \times Val \times Val \times Q_s \times (markedVal(\mathcal{M}, \mathcal{D}_\mathcal{L}, n) \cup \{\epsilon\}) \times \{0, \ldots, k\}$ is the set of states. A configuration $(q, d_c, d_g, q_s, mak, cnt) \in Q_1$ consists of a control state q, a valuation d_c of the memory, a valuation d_g of the memory which is generated from d_c by applying all the stored items in c_i, a state q_s for monitoring whether the extended history *eh* happens, a marker *mak* which is used to ensure that each specific $cas(n\text{+}1, z_w, \_, \_)$ action is immediately followed by the corresponding call or return action, and the number *cnt* of the call, return, flushCall and flushReturn actions already occurred in the whole trace.

Construction of Σ: $\Sigma = \Sigma_{s1} \cup \Sigma_{s2} \cup \Sigma_{s3}$ is the alphabet of channel contents with $\Sigma_{s1} = \{(n\text{+}1, z_w, d) | d \in Val\} \cup \{(i, z_f, d) | 1 \leq i \leq n, d \in Val\}$, $\Sigma_{s2} = \{((i, x, d), \sharp) | 1 \leq i \leq n, x \in \mathcal{X}_\mathcal{L}, d \in Val\}$ and $\Sigma_{s3} = \{a | (a, \sharp) \in \Sigma_{s2}\}$. Items in Σ_{s1} are inserted by guessing the specific *cas* actions of z_w or guessing *write* actions of z_f. Items in Σ_{s2} are either the newest item in c_i or the newest item for some memory location of $\mathcal{X}_\mathcal{L}$ in c_i. Items in Σ_{s3} are inserted by write or guess write actions. When M_i^k is considered as a lossy channel machine, Σ_{s1} and Σ_{s2} are the sets of strong symbols.

Construction of Λ: Λ is the set of transition labels and is union of the following sets:

- $\{write(i,x,d), cas(i,x,d_1,d_2) | d, d_1, d_2 \in Val, (1 \leq i \leq n \wedge x \in \mathcal{X}_\mathcal{L} \cup \{z_f\}) \vee (i = n\text{+}1 \wedge x = z_w)\}$.
- $\{flush(i,x,d), flush(n\text{+}1, z_w, d) | 1 \leq i \leq n, x \in \mathcal{X}_\mathcal{L} \cup \{z_f\}, d \in Val\}$.
- $\{call(i,m,a), return(i,m,a), flushCall(i), flushReturn(i) | 1 \leq i \leq n, m \in \mathcal{M}, a \in \mathcal{D}_\mathcal{L}\}$.

Λ does not contain read or τ actions, which are seen as ϵ transition in M_i^k.

Construction of Δ_i^k: Δ_i^k is the transition relation of M_i^k. Δ_i^k is the smallest set of transitions such that $\forall q \in Q_c \cup (Q_\mathcal{L} \times Q_c)$, $q_1, q_2 \in Q_\mathcal{L}$, $d_c, d_g \in Val$, $q_s \in Q_s$ and $cnt < k$,

- Nop: We only change the control state. Formally, if $q_1 \xrightarrow{\tau}'_\mathcal{L} q_2$, then $((q_1, q'_c), d_c, d_g, q_s, \epsilon, cnt) \xrightarrow{\epsilon, c_i : \Sigma, nop}_{\Delta_i^k} ((q_2, q'_c), d_c, d_g, q_s, \epsilon, cnt)$.
- Write by Library to $x \in \mathcal{X}_\mathcal{L}$: if $q_1 \xrightarrow{write(x,a)}'_\mathcal{L} q_2$ and $x \in \mathcal{X}_\mathcal{L}$, then

$$((q_1, q_c'), d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,(\beta_1,\sharp)\in c_i \wedge (\beta_2,\sharp)\in c_i, c_i[\beta_1/(\beta_1,\sharp),\beta_2/(\beta_2,\sharp)]!\beta_3}_{\Delta_i^k} ((q_2, q_c'), d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

$$((q_1, q_c'), d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,(\beta_1,\sharp)\in c_i \wedge c_i:\Theta_2, c_i[\beta_1/(\beta_1,\sharp)]!\beta_3}_{\Delta_i^k} ((q_2, q_c'), d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

$$((q_1, q_c'), d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,c_i:\Theta_1 \wedge (\beta_2,\sharp), c_i[\beta_2/(\beta_2,\sharp)]!\beta_3}_{\Delta_i^k} ((q_2, q_c'), d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

$$((q_1, q_c'), d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,c_i:\Theta_1 \wedge c_i:\Theta_2, c_i!\beta_3}_{\Delta_i^k} ((q_2, q_c'), d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

where $\beta_1 = (i, x, d_1) \wedge d_1 \in \mathit{Val}$, $\Theta_1 = \Sigma \backslash \{((i, x, d'), \sharp) | d' \in \mathit{Val}\}$, $\beta_2 = (j, \_, d_2)$ with $1 \le j \le n \wedge j \ne i \wedge d_2 \in \mathit{Val}$, $\Theta_2 = \Sigma \backslash \{((j, \_, d'), \sharp) | 1 \le j \le n, j \ne i, d' \in \mathit{Val}\}$, $d_g' = d_g[x : a]$, $\beta_3 = ((i, x, d_g'), \sharp)$ and $op = \mathit{write}(i, x, d_g')$.

- Write by Library to z_f: if $\mathit{is}_{(m,a)} \xrightarrow{\mathit{write}(z_f,\mathit{call})}{}'_{\mathcal{L}} q_2$, then

$$((\mathit{is}_{(m,a)}, q_c'), d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,(\beta_1,\sharp)\in c_i, c_i[\beta_1/(\beta_1,\sharp)]!\beta_2}_{\Delta_i^k} ((q_2, q_c'), d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

$$((\mathit{is}_{(m,a)}, q_c'), d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,c_i:\Theta_1, c_i!\beta_2}_{\Delta_i^k} ((q_2, q_c'), d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

where $\beta_1 = (j, \_, d)$ with $1 \le j \le n \wedge j \ne i \wedge d \in \mathit{Val}$, $\Theta_1 = \Sigma \backslash \{((j, \_, d'), \sharp) | 1 \le j \le n, j \ne i, d' \in \mathit{Val}\}$, $d_g' = d_g[z_f : \mathit{call}]$, $\beta_2 = (i, z_f, d_g')$ and $op = \mathit{write}(i, z_f, d_g')$.

- Write by Client to z_f:

$$(q_c'', d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,(\beta_1,\sharp)\in c_i, c_i[\beta_1/(\beta_1,\sharp)]!\beta_2}_{\Delta_i^k} (q_c, d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

$$(q_c'', d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,c_i:\Theta_1, c_i!\beta_2}_{\Delta_i^k} (q_c, d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

where $\beta_1 = (j, \_, d)$ with $1 \le j \le n \wedge j \ne i \wedge d \in \mathit{Val}$, $\Theta_1 = \Sigma \backslash \{((j, \_, d'), \sharp) | 1 \le j \le n, j \ne i, d' \in \mathit{Val}\}$, $d_g' = d_g[z_f : \mathit{ret}]$, $\beta_2 = (i, z_f, d_g')$ and $op = \mathit{write}(i, z_f, d_g')$.

- Guess Write to $x \in \mathcal{X}_{\mathcal{L}}$: if $1 \le j \le n \wedge j \ne i \wedge x \in \mathcal{X}_{\mathcal{L}} \wedge a \in \mathcal{D}_{\mathcal{L}}$, then

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,(\beta,\sharp)\in c_i, c_i[\beta/(\beta,\sharp)]!\beta'}_{\Delta_i^k} (q, d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,c_i:\Theta, c_i!\beta'}_{\Delta_i^k} (q, d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

where $\beta = (j_1, \_, \_)$ with $1 \le j_1 \le n \wedge j_1 \ne i$, $d_g' = d_g[x : a]$, $\beta' = ((j, x, d_g'), \sharp)$, $\Theta = \Sigma \backslash \{((j_2, \_, \_), \sharp) | 1 \le j_2 \le n \wedge j_2 \ne i\}$ and $op = \mathit{write}(j, x, d_g')$.

- Guess Write to z_w: if $a \in \mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n)$, then

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,(\beta,\sharp)\in c_i, c_i[\beta/(\beta,\sharp)]!\beta'}_{\Delta_i^k} (q, d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,c_i:\Theta, c_i!\beta'}_{\Delta_i^k} (q, d_c, d_g', q_s, \epsilon, \mathit{cnt})$$

where $\beta = (j, \_, \_)$ with $1 \le j \le n \wedge j \ne i$, $d_g' = d_g[z_w : a]$, $\beta' = (n{+}1, z_w, d_g')$, $\Theta = \Sigma \backslash \{((j, \_, \_), \sharp) | 1 \le j \le n \wedge j \ne i\}$ and $op = \mathit{write}(n{+}1, z_w, d_g')$.

- Guess Write to z_f: if $1 \leq j \leq n \wedge j \neq i \wedge a \in \{\mathit{call}, \mathit{ret}\}$, then

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op,(\beta,\sharp)\in c_i, c_i[\beta/(\beta,\sharp)]!\beta'}_{\Delta_i^k} (q, d_c, d'_g, q_s, \epsilon, \mathit{cnt})$$

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op, c_i:\Theta, c_i!\beta'}_{\Delta_i^k} (q, d_c, d'_g, q_s, \epsilon, \mathit{cnt})$$

where $\beta = (j', \_, \_)$ with $1 \leq j' \leq n \wedge j' \neq i$, $d'_g = d_g[z_f : a]$, $\beta' = (j, z_f, d'_g)$, $\Theta = \Sigma \backslash \{((j', \_, \_), \sharp) | 1 \leq j' \leq n \wedge j' \neq i\}$ and $op = \mathit{write}(j, z_f, d'_g)$.

- Library read: if $q_1 \xrightarrow{read(x,a)}{}'_{\mathcal{L}} q_2$, then for each $d \in \mathit{Val}$ with $d(x) = a$,

$$((q_1, q'_c), d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{\epsilon,(\beta,\sharp)\in c_i, nop}_{\Delta_i^k} ((q_2, q'_c), d_c, d_g, q_s, \epsilon, \mathit{cnt})$$

$$((q_1, q'_c), d, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{\epsilon, c_i:\Theta, nop}_{\Delta_i^k} ((q_2, q'_c), d, d_g, q_s, \epsilon, \mathit{cnt})$$

where $\beta = (i, x, d)$ and $\Theta = \Sigma \backslash \{((i, x, d'), \sharp) | d' \in \mathit{Val}\}$.

- Library *cas*: if $q_1 \xrightarrow{cas\_suc(x,a,b)}{}'_{\mathcal{L}} q_2$, then for each $d \in \mathit{Val}$ with $d(x) = a$,

$$((q_1, q'_c), d, d, q_s, \epsilon, \mathit{cnt}) \xrightarrow{cas(i,x,d[x:b]), c_i=\epsilon, nop}_{\Delta_i^k} ((q_2, q'_c), d[x : b], d[x : b], q_s, \epsilon, \mathit{cnt})$$

If $q_1 \xrightarrow{cas\_fail(x,a,b)}{}'_{\mathcal{L}} q_2$, then for each $d \in \mathit{Val}$ with $d(x) \neq a$,

$$((q_3, q'_c), d, d, q_s, \epsilon, \mathit{cnt}) \xrightarrow{cas(i,x,d), c_i=\epsilon, nop}_{\Delta_i^k} ((q_4, q'_c), d, d, q_s, \epsilon, \mathit{cnt})$$

- Flush items of $x \in \mathcal{X}_{\mathcal{L}}$: for each $1 \leq j \leq n$, $x \in \mathcal{X}_{\mathcal{L}}$ and $d \in \mathit{Val}$,

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}') \xrightarrow{op, c_i:\Sigma, c_i?(j,x,d)}_{\Delta_i^k} (q, d, d_g, q_s, \epsilon, \mathit{cnt}')$$

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}') \xrightarrow{op, c_i:\Sigma, c_i?((j,x,d),\sharp)}_{\Delta_i^k} (q, d, d_g, q_s, \epsilon, \mathit{cnt}')$$

where $op = \mathit{flush}(j, x, d)$ and $\mathit{cnt}' \leq k$.

- Flush z_w Item When It Is a Call Item:

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op, c_i:\Sigma, c_i?(n+1, z_w, d)}_{\Delta_i^k} (q, d, d_g, q_s, \mathit{call}(j, m, c), \mathit{cnt})$$

where $d(z_w) = \mathit{call}(j, m, c)$ and $op = \mathit{flush}(n+1, z_w, d)$.

- Flush z_w Item When It Is a Return Item:

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op, c_i:\Sigma, c_i?(n+1, z_w, d)}_{\Delta_i^k} (q, d, d_g, q_s, \mathit{return}(j, m, c), \mathit{cnt})$$

where $d(z_w) = \mathit{return}(j, m, c)$ and $op = \mathit{flush}(n+1, z_w, d)$.

- Flush z_f Item When it is a Call Marker (flushCall): for each $1 \leq j \leq n$, if $q_s \xrightarrow{flushCall(j)}_s q'_s$, then

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op, c_i:\Sigma, c_i?(j, z_f, d)}_{\Delta_i^k} (q, d, d_g, q'_s, \epsilon, \mathit{cnt}+1)$$

where $d(z_f) = \mathit{call}$ and $op = \mathit{flushCall}(j)$.

- Flush z_f Item When it is a Return Marker (flushReturn): for each $1 \leq j \leq n$, if $q_s \xrightarrow{\mathit{flushReturn}(j)}_s q'_s$, then

$$(q, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op, c_i:\Sigma, c_i?(j, z_f, d)}_{\Delta_i^k} (q, d, d_g, q'_s, \epsilon, \mathit{cnt+1})$$

where $d(z_f) = \mathit{ret}$ and $op = \mathit{flushReturn}(j)$.
- Call: if $q_s \xrightarrow{\mathit{call}(i,m,a)}_s q'_s$, then

$$(q_c, d_c, d_g, q_s, \mathit{call}(i, m, a), \mathit{cnt}) \xrightarrow{\mathit{call}(i,m,a), c_i:\Sigma, \mathit{nop}}_{\Delta_i^k} ((\mathit{is}_{(m,a)}, q'_c), d_c, d_g, q'_s, \epsilon, \mathit{cnt+1})$$

- Guess Call: if $q_s \xrightarrow{\mathit{call}(j,m,a)}_s q'_s$, $1 \leq j \leq n$ and $j \neq i$, then

$$(q, d_c, d_g, q_s, \mathit{call}(j, m, a), \mathit{cnt}) \xrightarrow{\mathit{call}(j,m,a), c_i:\Sigma, \mathit{nop}}_{\Delta_i^k} (q, d_c, d_g, q'_s, \epsilon, \mathit{cnt+1})$$

- Return: if $q_s \xrightarrow{\mathit{return}(i,m,a)}_s q'_s$, then

$$((\mathit{fs}_{(m,a)}, q'_c), d_c, d_g, q_s, \mathit{return}(i, m, a), \mathit{cnt}) \xrightarrow{\mathit{return}(i,m,a), c_i:\Sigma, \mathit{nop}}_{\Delta_i^k} (q''_c, d_c, d_g, q'_s, \epsilon, \mathit{cnt+1})$$

- Guess Return: if $q_s \xrightarrow{\mathit{return}(j,m,a)}_s q'_s$, $1 \leq j \leq n$ and $j \neq i$, then

$$(q, d_c, d_g, q_s, \mathit{return}(j, m, a), \mathit{cnt}) \xrightarrow{\mathit{return}(j,m,a), c_i:\Sigma, \mathit{nop}}_{\Delta_i^k} (q, d_c, d_g, q'_s, \epsilon, \mathit{cnt+1})$$

We intuitively explain several typical transition relations of M_i^k. For writing z_f by library, we need to erase the $\sharp$ symbol in the original newest item of c_i (if it is inserted by guessing write of process $j \neq i$) if it exists. Then we insert a z_f item of *call* into c_i. For guessing write actions of process *n+1* to z_w, we need to erase the $\sharp$ symbol in the original newest item of c_i (if it is inserted by guessing write of process $j \neq i$) if it exists. Then we insert a z_w item of $\mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n)$ into c_i. If the *mak* tuple is a call action of process $j \neq i$, then we can launch a same call action, change the q_s tuple, unset *mak* tuple to ϵ and increase the *cnt* tuple. If the *mak* tuple is a return action of process i and the control state of process i can launch a return action, then we can launch a same return action, change the q_s tuple, change the control state of process i, unset *mak* tuple to ϵ and increase the *cnt* tuple.

7.3 Construction of M_{n+1}^k

In this subsection, we present the formal definition of the channel machine M_{n+1}^k, which simulates the behavior of process *n+1* of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n+1]\!]_b$ and monitors whether a specific k-extended history happens.

Channel machine M_{n+1}^k is a tuple $(Q_{n+1}^k, \{c_{n+1}\}, \Sigma, \Lambda, \Delta_{n+1}^k)$, where Q_{n+1}, c_{n+1} and Δ_{n+1}^k are defined as follows:

$Q^k_{n+1} = \{q_{wit}\} \times \mathit{Val} \times \mathit{Val} \times Q_s \times (\mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n) \cup \{\epsilon\}) \times \{0, \ldots, k\}$ is the set of states. Recall that, as stated in Section 6, q_{wit} is the control state of the client program C_{marked}, which is used by process *n+1*.

c_{n+1} is name of the single channel of M^k_{n+1}.

Δ^k_{n+1} is the transition relation of M^k_{n+1}. Here we only explain the *cas* transitions, and other transitions of Δ^k_{n+1} is similar to Δ^k_i, and the main difference is that the control state is fixed to be q_{wit}. Δ^k_{n+1} is the smallest set of transitions such that $\forall d_c, d_g \in \mathit{Val}$, $q_s \in Q_s$ and $cnt < k$,

- Client *cas*: The *cas* actions in C_{marked} are for z_w. We need the channel to be cleared and require $d_c = d_g$. Then we change d_c and d_g directly. Formally, for each $b \in \mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n)$ and $d \in \mathit{Val}$, then

$$(q_{wit}, d, d, q_s, \epsilon, cnt) \xrightarrow{cas(i,x,d[x_{wit}:b]),c_i=\epsilon,nop}_{\Delta^{ts}_i} (q_{wit}, d[x_{wit} : b], d[x_{wit} : b], q_s, b, cnt)$$

- Guess write to $x \in \mathcal{X}_{\mathcal{L}}$: if $1 \leq j \leq n \wedge x \in \mathcal{X}_{\mathcal{L}} \wedge a \in \mathcal{D}_{\mathcal{L}}$, then

$$(q_{wit}, d_c, d_g, q_s, \epsilon, cnt) \xrightarrow{op,(\beta,\sharp)\in c_i,c_i[\beta/(\beta,\sharp)]!\beta'}_{\Delta^{ts}_i} (q_{wit}, d_c, d'_g, q_s, \epsilon, cnt)$$

$$(q_{wit}, d_c, d_g, q_s, \epsilon, cnt) \xrightarrow{op,c_i:\Theta,c_i!\beta'}_{\Delta^{ts}_i} (q_{wit}, d_c, d'_g, q_s, \epsilon, cnt)$$

where $\beta = (j_1, \_, \_)$ with $1 \leq j_1 \leq n$, $d'_g = d_g[x : a]$, $\beta' = ((j, x, d'_g), \sharp)$, $\Theta = \Sigma \backslash \{((j_1, \_, \_), \sharp) | 1 \leq j_1 \leq n\}$ and $op = \mathit{write}(j, x, d'_g)$.
- Guess write to z_f: for each $a \in \{\mathit{call}, \mathit{ret}\}$ and $d \in \mathit{Val}$

$$(q_{wit}, d_c, d_g, q_s, \epsilon, cnt) \xrightarrow{op,(\beta_1,\sharp)\in c_i,c_i[\beta_1/(\beta_1,\sharp)]!\beta_2}_{\Delta^k_i} (q_{wit}, d_c, d'_g, q_s, \epsilon, cnt)$$

$$(q_{wit}, d_c, d_g, q_s, \epsilon, cnt) \xrightarrow{op,c_i:\Theta_1,c_i!\beta_2}_{\Delta^k_i} (q_{wit}, d_c, d'_g, q_s, \epsilon, cnt)$$

where $\beta_1 = (j, \_, d)$ with $1 \leq j \leq n$, $\Theta_1 = \Sigma \backslash \{((j, \_, d'), \sharp) | 1 \leq j \leq n, d' \in \mathit{Val}\}$, $d'_g = d_g[z_f : a]$, $\beta_2 = (i, z_f, d'_g)$ and $op = \mathit{write}(i, z_f, d'_g)$.
- Flush items of $x \in \mathcal{X}_{\mathcal{L}}$: if $1 \leq j \leq n$, then for each $x \in \mathcal{D}_{\mathcal{L}}$, $d \in \mathit{Val}$,

$$(q_{wit}, d_c, d_g, q_s, \epsilon, cnt') \xrightarrow{op,c_i:\Sigma,c_i?(j,x,d)}_{\Delta^{ts}_i} (q_{wit}, d, d_g, q_s, \epsilon, cnt')$$

$$(q_{wit}, d_c, d_g, q_s, \epsilon, cnt') \xrightarrow{op,c_i:\Sigma,c_i?((j,x,d),\sharp)}_{\Delta^{ts}_i} (q_{wit}, d, d_g, q_s, \epsilon, cnt')$$

where $op = \mathit{flush}(j, x, d)$ and $cnt' \leq k$ and .
- Guess call: if $q_s \xrightarrow{call(j,m,a)}_s q'_s$ and $1 \leq j \leq n$, then

$$(q_{wit}, d_c, d_g, q_s, \mathit{call}(j, m, a), cnt) \xrightarrow{call(j,m,a),c_i:\Sigma,nop}_{\Delta^{ts}_i} (q_{wit}, d_c, d_g, q'_s, \epsilon, cnt{+}1)$$

- Guess return: if $q_s \xrightarrow{return(j,m,a)}_s q'_s$ and $1 \leq j \leq n$, then

$$(q_{wit}, d_c, d_g, q_s, \mathit{return}(j, m, a), cnt) \xrightarrow{return(j,m,a),c_i:\Sigma,nop}_{\Delta^{ts}_i} (q_{wit}, d_c, d_g, q'_s, \epsilon, cnt{+}1)$$

- Flush z_f Item When it is a Call Marker (flushCall): if $q_s \xrightarrow{\mathit{flushCall}(j)}_s q'_s$ and $1 \leq j \leq n$, then

$$(q_{wit}, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op, c_i:\Sigma, c_i?(j, z_f, d)}_{\Delta_i^k} (q_{wit}, d, d_g, q'_s, \epsilon, \mathit{cnt}+1)$$

where $d(z_f) = \mathit{call}$ and $op = \mathit{flushCall}(j)$.

- Flush z_f Item When it is a Return Marker (flushReturn): if $q_s \xrightarrow{\mathit{flushReturn}(j)}_s q'_s$ and $1 \leq j \leq n$, then

$$(q_{wit}, d_c, d_g, q_s, \epsilon, \mathit{cnt}) \xrightarrow{op, c_i:\Sigma, c_i?(j, z_f, d)}_{\Delta_i^k} (q_{wit}, d, d_g, q'_s, \epsilon, \mathit{cnt}+1)$$

where $d(z_f) = \mathit{return}(m, c)$ and $op = \mathit{flushReturn}(j)$.

8 Reducing *k*-Bounded TSO-to-TSO Linearizability to Channel Machines

In this section, we prove that k-bounded TSO-to-TSO linearizability is decidable by reducing it into several control state reachability problems of (S, K)-channel machines.

Let $M_i^{k\text{-}(f,exth)}$ (resp., $M_i^{k\text{-}w}$) be a channel machine that is generated from M_i^k by replacing all but but flush, *cas*, call, return, flushCall and flushReturn transitions (resp., all but write and *cas* transitions) with internal transitions, and replacing each $\mathit{cas}(\mathit{ind}, x, d_1, d_2)$ transition with $\mathit{flush}(\mathit{ind}, x, d_2)$ transition (resp., $\mathit{write}(\mathit{ind}, x, d_2)$ transition) for each $\mathit{ind}, x, d_1, d_2$. Let $M_i^{k\text{-}f}$ be a channel machine that is generated from M_i^k by (1) replacing all but flush, *cas*, flushCall and flushReturn transitions with internal transitions, (2) replacing each $\mathit{cas}(\mathit{ind}, x, d_1, d_2)$ transition with $\mathit{flush}(\mathit{ind}, x, d_2)$ transition, and (3) replacing each $\mathit{flushCall}(\mathit{ind})$ transition (resp., $\mathit{flushReturn}(\mathit{ind})$ transition) with $\mathit{flush}(\mathit{ind}, z_f, d)$ transition where d is the valuation of the flushed item and $d(z_f) = \mathit{call}$ (resp., $d(z_f) = \mathit{ret}$).

To deal with k-extended histories, we need to consider only the paths which contains at most k call, return, flushCall and flushReturn actions, and at every state of the path, the number of strong symbol in Σ_{s2} is always less or equal than $|\mathcal{X}|$+1 (possibly one for the newest item in channel, and the number for newest items of each memory location is at most $|\mathcal{X}|$). Therefore, Let $S = \langle \Sigma_1, \Sigma_2 \rangle$ be the sets of strong symbols, and let $K_1 = \langle k, |\mathcal{X}_{\mathcal{L}}|+1 \rangle$ specify the maximal permitted number of strong symbols in each set.

The following lemma states that the existence of a marked witness from a specific begin state to a specific end state of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n+1]\!]_b$ implies a control state reachability problem of a channel machine which is the production of $M_1^{k\text{-}(f,exth)}$ to $M_{n+1}^{k\text{-}(f,exth)}$ (generated with the specific begin state and end state for each process). We formally prove this lemma in Appendix A.2. Here p_{in} maps $1 \leq i \leq n$ into q_c and maps $n+1$ into q_{wit}.

Lemma 5. *Given a k-extended history eh. If there exists a marked witness t of eh from $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, \epsilon)$ in $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n+1]\!]_b$, then $\cap_{i=1}^{n+1} T_{(q_i, q'_i)}^{(S,K_1)} M_i^{k\text{-}(f,exth)} \neq \emptyset$, where for each $1 \leq i \leq n+1$, $q_i = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon, 0)$, $q'_i = (p_w(i), d_w, d_w, q_{ends}, \epsilon, |t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})}|)$.*

Proof. (Intuition and Sketch)

To prove this lemma, given a path pa_b of t in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, we construct a path pa_m of $M_1^{k\text{-}(f,exth)} \otimes M_{n+1}^{k\text{-}(f,exth)}$, where pa_b and pa_m has same extended history.

If we can find a weak simulation relation or something alike between states of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$ and states of $M_1^{k\text{-}(f,exth)} \otimes M_{n+1}^{k\text{-}(f,exth)}$, then this lemma can be proved. However, this is hard, since in a state s_b of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, the content of buffers can be flushed out in many orders, while in a state s_m of $M_1^{k\text{-}(f,exth)} \otimes M_{n+1}^{k\text{-}(f,exth)}$, each $M_i^{k\text{-}(f,exth)}$ see a same total store order, and thus, the content in channel can be flushed out in an unique order. This implies that s_b has "more possibility future behaviors" than s_m to some extent.

To solve this problem, we introduce an intermediate operational semantics $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$. A state s_g of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$ is generated from s_b by adding a total store order of some execution, and we force that the execution that contains s_g must act according to the total store. Since s_g restrict behavior of s_b with total store order, we can now relate s_g and s_m with some weak simulation relation.

Given a path pa_b in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, it is easy to construct a path pa_g in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$ by adding the total store order of pa_b into each state of pa_b. Then we construct a weak simulation relation between states of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$ and states of $M_1^{k\text{-}(f,exth)} \otimes M_{n+1}^{k\text{-}(f,exth)}$, which implies that k-extended histories of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$ is a subset of extended histories of $M_1^{k\text{-}(f,exth)} \otimes M_{n+1}^{k\text{-}(f,exth)}$. This completes the proof of this lemma. □

The following lemma states that a control state reachability problem of a channel machine which is the production of $M_1^{k\text{-}(f,exth)}$ to $M_{n+1}^{k\text{-}(f,exth)}$ (generated with the specific begin state and end state for each process) implies the existence of a marked witness from a specific begin state to a specific end state of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$. We formally prove it in Appendix A.3.

Lemma 6. *Given a k-extended history eh. If $\cap_{i=1}^{n+1} T_{(q_i,q_i')}^{(S,K_1)} M_i^{k\text{-}(f,exth)} \neq \emptyset$, where for each $1 \le i \le n{+}1$, $q_i = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon, 0)$, $q_i' = (p_w(i), d_w, d_w, q_{ends}, \epsilon, |eh|)$, then there exists a marked witness t of eh from $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, \epsilon)$ in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$.*

Proof. (Sketch)

We construct a $\sim$ relation between states of $M_1^{k\text{-}(f,exth)} \otimes \ldots \otimes M_{n+1}^{k\text{-}(f,exth)}$ and states of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, such that if $q_1 \sim q_1'$ and $q_1 \xrightarrow{act} q_2$, then one of the following two cases holds:

- q_1' can launch $act_1' \ldots act_l'$ transitions to a state q_2', such that $q_2 \sim q_2'$, and the call, return, flushCall and flushReturn actions in *act* are same to that in $act_1' \ldots act_l'$.
- $q_2 \sim q_1'$.

Then it is obvious that from a path pa_m of $M_1^{k\text{-}(f,exth)} \otimes \ldots \otimes M_{n+1}^{k\text{-}(f,exth)}$, we can generate a path pa_b of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, such that pa_m and pa_b has the same call, return, flushCall and flushReturn actions. □

Let Σ_f be the set of flush actions. Given a sequence l of flush, call, return, flushCall and flushReturn actions, let *FcrToF*(l) be a set of sequences, each of which is generated from l by (1) erasing call and return actions, (2) for each i, transforming *flushCall*(i) and *flushReturn*(i) actions into *flush*(i, z_f, d_1) and *flush*(i, z_f, d_2) actions for some $d_1, d_2 \in$ *Val*, where $d_1(z_f) =$ *call* and $d_2(z_f) =$ *ret*. The following lemma states that a control state reachability problem of the channel machine $M_i^{\textit{k-(f,exth)}}$ is equivalent to a control state reachability problem of the channel machine $M_i^{\textit{k-f}}$.

Lemma 7. *Let l be a sequence of flush actions. Then, $l \in T_{(q,q')}^{(S,K_1)} M_i^{\textit{k-f}}$, if and only if there exists a sequence l' of flush, call, return, flushCall and flushReturn actions, such that $l' \in T_{(q,q')}^{(S,K_1)} M_i^{\textit{k-(f,exth)}}$ and FcrToF$(l') = l$. Here $(q = (q_c, d_{init}, d_{init}, q_{is}, \epsilon, 0) \wedge 1 \leq i \leq n) \vee (q = (q_{wit}, d_{init}, d_{init}, q_{is}, \epsilon, 0) \wedge i = n{+}1)$, and $q' = (\_, d, d, q_{ends}, \epsilon, |l' \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})} |)$ for some $d \in$ Val.*

Proof. The *if* direction is obvious. To prove the *only if* direction, we generate a sequence l'' from l' as follows: Since $M_i^{\textit{k-f}}$ ensures that each *flush*$(\_, z_w, \_)$ action is immediately followed by a corresponding call or return action, we add the corresponding call and return actions right after the *flush*$(\_, z_w, \_)$ actions. Then we transform each *flush*(i, z_f, d_1) and *flush*(i, z_f, d_2) actions into *flushCall*(i) and *flushReturn*(i) actions, where $1 \leq i \leq n$, $d_1, d_2 \in$ *Val*, $d_1(z_f) =$ *call* and $d_2(z_f) =$ *ret*. It is easy to see that $l'' \in T_{(q,q')}^{(S,K_1)} M_i^{\textit{k-(f,exth)}}$. □

Given a finite sequence l of flush actions, let *Trans*$_{f \to w}(l)$ be a finite sequence that is generated from l by transforming each *flush*(i, x, d) action to *write*(i, x, d) action. The following lemma states that a control state reachability problem of the channel machine $M_i^{\textit{k-f}}$ is equivalent to a control state reachability problem of the channel machine $M_i^{\textit{k-w}}$.

Lemma 8. *Let l be a sequence of flush actions. $l \in T_{(q,q')}^{(S,K_1)} M_i^{\textit{k-f}}$ if and only if Trans$_{f \to w}(l) \in T_{(q,q')}^{(S,K_1)} M_i^{\textit{k-w}}$, where $(q = (q_c, d_{init}, d_{init}, q_{is}, \epsilon, 0) \wedge 1 \leq i \leq n) \vee (q = (q_{wit}, d_{init}, d_{init}, q_{is}, \epsilon, 0) \wedge i = n{+}1)$ and $q' = (\_, d, d, q_{ends}, \epsilon, a)$ for some $d \in$ Val, and a is the number of flush action of z_w and z_f in l.*

Proof. This lemma holds because that, in a perfect channel machine, the sequences of values put into the channels is always equal to the sequences of values take out from the channels. □

Based on Lemma 5, Lemma 6, Lemma 7 and Lemma 8, we can reduce the existence of a marked witness from a specific begin state to a specific end state of $[\![\textit{Clt}_f(\textit{Mod}(\mathcal{L})), n{+}1]\!]_b$ into a control state reachability problem of a channel machine which is the production of $M_1^{\textit{k-w}}$ to $M_{n+1}^{\textit{k-w}}$ (generated with the specific begin state and end state for each process).

Lemma 9. *Given a k-extended history eh. There exists a marked witness t of eh from $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, \epsilon)$ in $[\![\textit{Clt}_f(\textit{Mod}(\mathcal{L})), n{+}1]\!]_b$, if and only if $\cap_{i=1}^{n+1} T_{(q_i,q_i')}^{(S,K_1)} M_i^{\textit{k-w}} \neq \emptyset$, where for each $1 \leq i \leq n{+}1$, $q_i = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon, 0)$, $q_i' = (p_w(i), d_w, d_w, q_{ends}, \epsilon, |t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})} |)$.*

Proof. Obvious from Lemma 5, Lemma 6, Lemma 7 and Lemma 8. $\square$

The following lemma states that we can strengthen the result of Lemma 9 to the situation when $M_1^{k\text{-}w}, \dots, M_{n+1}^{k\text{-}w}$ are interpreted as lossy channel machines. We formally prove this lemma in Appendix A.4.

Lemma 10. *Given a k-extended history eh. There exists a marked witness t of eh from $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, \epsilon)$ in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, if and only if $\cap_{i=1}^{n+1} LT_{(q_i,q_i')}^{(S,K_1)} M_i^{k\text{-}w} \neq \emptyset$, where for each $1 \le i \le n{+}1$, $q_i = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon, 0)$, $q_i' = (p_w(i), d_w, d_w, q_{ends}, \epsilon, |t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})}|)$.*

Proof. (Sketch)
We first prove the fact that if $(s_1, c_1) \xrightarrow{act}_{(M_i^{k\text{-}w}, S, K_1)} (s_2, c_2)$ and $c_1 \preceq_S^{K_1} c_1'$, then there exists c_2' and β, such that $c_2 \preceq_S^{K_1} c_2'$, $(s_1, c_1') \xrightarrow{act_1'}_{M_i^{k\text{-}w}} \dots \xrightarrow{act_l}_{M_i^{k\text{-}w}} (s_1, c_2')$, and write actions in *act* equals to write actions in $act_1 \cdot \ldots \cdot act_l$.

According to Lemma 9, we reduce the existence of t into $\cap_{i=1}^{n+1} T_{(q_i,q_i')}^{(S,K_1)} M_i^{k\text{-}w} \neq \emptyset$. Then, we need to prove that, $\cap_{i=1}^{n+1} T_{(q_i,q_i')}^{(S,K_1)} M_i^{k\text{-}w} \neq \emptyset$, if and only if $\cap_{i=1}^{n+1} LT_{(q_i,q_i')}^{(S,K_1)} M_i^{k\text{-}w} \neq \emptyset$. The *only if* direction is obvious, since a sequence $l \in \cap_{i=1}^{n+1} T_{(q_i,q_i')}^{(S,K_1)} M_i^{k\text{-}w}$ is also a sequence of $\cap_{i=1}^{n+1} LT_{(q_i,q_i')}^{(S,K_1)} M_i^{k\text{-}w}$. To prove the *if* direction, with the fact proved in this lemma, we can construct a path of perfect channel from a path of lossy channel step by step, where both path has same write actions. This completes the proof of this lemma. $\square$

The following theorem states that k-bounded TSO-to-TSO linearizability is decidable. By enumerating all possible start states and end states, and repeatedly apply Lemma 10, we can check whether a specific k-extended history *eh* is in *ehistory*$([\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b)$. Since the number of k-extended histories is finite, by repeatedly apply above process, we can obtain *k-ehistory*$([\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b)$ and *k-ehistory*$([\![Clt_f(Mod(\mathcal{L}')), n{+}1]\!]_b)$. Then we can decide k-bounded TSO-to-TSO linearizability.

Theorem 1. *Given concurrent data structure $\mathcal{L}$ and $\mathcal{L}'$, the problem of whether $\mathcal{L}'$ k-bounded TSO-to-TSO linearizes $\mathcal{L}$ for n processes is decidable.*

Proof. There are only finite number of $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ and $(p_w, d_w, \epsilon^{n+1}, \epsilon)$. By applying Lemma 10 with all possible $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ and $(p_w, d_w, \epsilon^{n+1}, \epsilon)$, we reduce the problem of checking whether there exists a marked witness for *eh* into a finite number of problems of checking $\cap_{i=1}^{n+1} LT_{(\_,\_)}^{(S,K_1)} M_i^{k\text{-}w} \neq \emptyset$. By Lemma 3, this is reduced to several control state reachability problem sof (S,K)-*LCM*, which is known decidable.

By Lemma 4 and Lemma 2, the existence of specific extended history *eh* in *ehistory*$([\![\mathcal{L}, n]\!]_{tt})$ is equivalent to the existence of marked witness for *eh* in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$. Therefore, we can now decide whether a specific extended history *eh* is in *ehistory*$([\![\mathcal{L}, n]\!]_{tt})$.

Since the number of k-extended histories is bounded by n, $\mathcal{M}$ and $\mathcal{D}_{\mathcal{L}}$, we can obtain *k-ehistory*$([\![\mathcal{L}, n]\!]_{tt})$ by first enumerating all possible k-extended histories, and then use above process to check whether each k-extended history is in *k-ehistory*$([\![\mathcal{L}, n]\!]_{tt})$.

Similarly, we can obtain *k-ehistory*($[\![\mathcal{L}', n]\!]_{tt}$). Consider *k-ehistory*($[\![\mathcal{L}, n]\!]_{tt}$) and *k-ehistory*($[\![\mathcal{L}', n]\!]_{tt}$), by Lemma 1, it is decidable whether $\mathcal{L}'$ k-bounded TSO-to-TSO linearizes $\mathcal{L}$ for n processes. □

9 Complexity of k-Bounded TSO-to-TSO Linearizability

In this section, we prove that the TSO-to-TSO linearizability problem has non-primitive recursive complexity.

9.1 (Lossy) Simple channel machine

A simple channel machine [15] is a kind of channel machine that has only one channel and simple transition rules. Formally a channel machine $M = (Q, \mathcal{CH}, \Sigma_{\mathcal{CH}}, \Lambda, \Delta)$ is called a simple channel machine, if

- $\mathcal{CH}$ contains only one channel,
- each transition in $Delta$ uses a ϵ transition label, an empty guard, and does not use substitution,
- there is no strong symbol,

For simplicity, a simple channel machine M can be redefined as $M = (Q, \{c\}, \Sigma_c, \Delta_M)$, where Q is a finite set of states; c is the name of the only channel of M; Σ_c is the alphabet for channel contents; $\Delta_M \subseteq Q \times (\Sigma_{\mathcal{CH}} \cup \{\epsilon\}) \times (\Sigma_{\mathcal{CH}} \cup \{\epsilon\}) \times Q$ is the transition relation. A rule (q_1, u, v, q_2) is in Δ_M, if one of the following cases holds:

- there exists $(q_1, \epsilon, \epsilon, c?a, q_2) \in \Delta$, $u = a$ and $v = \epsilon$,
- there exists $(q_1, \epsilon, \epsilon, c!a, q_2) \in \Delta$, $u = \epsilon$ and $v = a$,
- there exists $(q_1, \epsilon, \epsilon, \mathit{nop}, q_2) \in \Delta$, $u = \epsilon$ and $v = \epsilon$

Intuitively, a transition rule (q_1, u, v, q_2) represents a transition from q_1 to q_2, which gets u (if $u \neq \epsilon$) from channel c and puts v (if $v \neq \epsilon$) into channel c. Given a lossy simple channel machine M and two configurations s_1, s_2 of M, the reachability problem of M is to determine whether there is a finite run from s_1 to s_2 in lossy semantics of M. According to [15], it is obvious that the reachability problem of lossy simple channel machine has nonprimitive recursive complexity.

9.2 Concurrent Data Structure $\mathcal{L}^{M}_{(s_1,s_2)}$

In this subsection, we construct concurrent data structure $\mathcal{L}^{M}_{(s_1,s_2)}$ to simulate the the reachability problem of lossy simple channel machine M from s_1 to s_2. This idea is inspired by our previous work in [13].

The buffer of TSO can contain unbounded number of items. A item can be putted into buffer by a write action, and when a item of x is flushed out from buffer, it can be detected by other processes by reading x. On TSO, flush actions are launched nondeterministically by the memory system. Therefore, between two consecutive read actions, more than one flush actions may happen. The next read action can only read the latest flush action, while missing the intermediate ones. These missing flush actions are similar to the missing messages that may happen in a lossy simple channel machine. However, a (simple) channel machine accesses the content of a channel always in an FIFO manner; while on the contrary, a process on the TSO memory model always reads the latest updates in its local store buffer (whenever possible). Therefore, we use two buffers of two processes to simulate one channel, where processes P_1 runs M_1 of $\mathcal{L}^M_{(s_1,s_2)}$ and process P_2 runs M_2 of $\mathcal{L}^M_{(s_1,s_2)}$. Process P_1 read updates of process P_2, change them according to transition rules of the lossy simple channel machine, and write them into buffer, while process P_2 read updates of process P_1 and write them into buffer. M_2 never return. M_1 does not return, until it finds that s_2 has been reached. In this way, each transition of the lossy simple channel machine can be reproduced through a round of communication of two processes, and the reachability problem of lossy simple channel machine is reduced into whether M_1 returns.

Formally, given a lossy simple channel machine $M = (Q, \{c\}, \Sigma_c, \Delta_M)$ and configurations $s_1 = (q_1, W_1), s_2 = (q_2, W_2)$ in the semantics of the lossy simple channel machine, $\mathcal{L}^M_{(s_1,s_2)}$ is constructed as follows: The finite data domain of $\mathcal{L}^M_{(s_1,s_2)}$ is $\mathcal{D}_{\mathcal{L}} = Q \cup \Sigma_c \cup \{\mathit{start}, \mathit{end}, \sharp, \bot, 0, \ldots, \mathit{max}(\{|W_1| + 1, \{|W_2| + 1\})\}$. $\mathcal{L}^M_{(s_1,s_2)}$ is constructed with two methods M_1 and M_2, and the following memory locations:

- a memory location x that is used to transmit the channel contents from M_1 to M_2,
- a memory location y that is used to transmit the channel contents from M_2 to M_1,
- a memory location *cnt* that is used in M_1 to count how many items has been read (we use $|W_2| + 1$ to represent the numbers larger or equal than $|W_2| + 1$) in current round.
- an array *RecvSeq* which is of length $|W_2|$ and is used to store the first $|W_2|$ items read in each round,

The symbol $\sharp$ is used as the delimiter to ensure that one element will not be read twice. The symbols *start* and *end* represent the start and the end of the channel contents, respectively. $\bot$ is used as the initial value of elements in *RecvSeq* in each round.

We now present the two methods in the pseudo-code, shown in Methods 1 and 2. For the sake of brevity, the following macro notations are used:

- For sequence $s = a_1 \cdot \ldots \cdot a_m$, we use *writeSeq(x,s)* to represent the commands of writing $a_1, \sharp, \ldots, a_m, \sharp$ to x in sequence,
- We use $v := \mathit{readOne}(x)$ to represent the commands of reading $e, \sharp$ from x in sequence for some $e \neq \sharp$ and then assigning e to v. We use $\mathit{readOneSpec}(x, v)$ to represent the commands of reading $a, \sharp$ from x in sequence where a is the value of v. If $\mathit{readOne}(x)$ reads $\sharp, \sharp$ from x, or $\mathit{readOneSpec}(x, v)$ fails to read the required value, then the calling process will no long proceed.
- We use $\mathit{writeOneSpec}(x, v)$ to represent the commands of writing a, $\sharp$ to x in sequence where a is the current value of v.
- We use $\mathit{initRecvSeq}()$ to represent the commands that assigns 0 to *cnt* and assigns $\bot$ to $\mathit{RecvSeq}(1), \ldots, \mathit{RecvSeq}(|W_2|)$.
- We use *det(q,cnt,ele)* to represent the macro, which add *ele* into *RecvSeq* and then determine whether *RecvSeq* equals W_2. It works as follows:

- If $q = q_2 \wedge W_2 = \epsilon$, then it returns *true*.
- Else, if $q \neq q_2 \vee \mathit{cnt} = |W_2|$, then it returns *false*.
- Else, if $0 \leq \mathit{cnt} < |W_2| - 1$, then it assigns *ele* to *RecvSeq*(cnt), increases *cnt* by 1 and returns *false*,
- Otherwise, $\mathit{cnt} = |W_2| - 1$ in this case. It assigns *ele* to *RecvSeq*$(|W_2| - 1)$, increases *cnt* by 1, and checks whether the contents of *RecvSeq* equals W_2. If the contents of *RecvSeq* equals W_2, then it returns *true*, otherwise, it increases *cnt* by 1 and returns *false*.

Method 1: M_1

```
   Input: an arbitrary argument
   Output: an arbitrary argument
1  writeSeq($x, q_1 \cdot start \cdot W_1 \cdot end$);
2  while true do
3      $q'_1 := readOne(y)$ for some state $q'_1 \in Q$;
4      guess a transition rule $rul = (q'_1, u, v, q'_2) \in \Delta_M$;
5      initRecvSeq();
6      if $q'_1 = q_2 \wedge W_2 = \epsilon$ then
7          return;
8      readOneSpec($y, start$);
9      if $u \neq \epsilon$ then
10         readOneSpec($y, u$);
11         if det($q'_1, cnt, u$) = true then
12             return;
13     writeSeq($x, q'_2 \cdot start$);
14     while true do
15         tmp = readOne(y);
16         if temp = end then
17             break;
18         writeOneSpec($x, tmp$);
19         if det($q'_1, cnt, tmp$) = true then
20             return;
21     if $v \neq \epsilon$ then
22         writeOneSpec($x, v$);
23     writeOneSpec($x, end$);
```

The pseudo-code of method M_1 is shown in Method 1. M_1 first puts $q_1 \cdot \mathit{start} \cdot W_1 \cdot \mathit{end}$ into the processor-local store buffer by writing them to x (Line 1). Then, it begins an infinite loop that never returns unless (q_2, W_2) is reached (Lines $2 - 23$). The round of Lines $2 - 23$ works as follows: It reads the current state q'_1 (Line 3) and guesses a transition rule $\mathit{rul} = (q'_1, u, v, q'_2) \in \Delta_M$ (Line 4). M_1 initializes *RecvSeq* (Line 5), check whether it is the case that $\mathit{tempQ} = q_2 \wedge W_2 = \epsilon$ (Lines $6 - 7$). If so, it returns as soon as possible. It not, it reads *start* from y (Line 8). If $u \neq \epsilon$, it read u from y (Lines 9-12). It writes $q'_2 \cdot \mathit{start}$ into x (Line 13). Then, it reads the remaining contents of "channel" (intermediate values of y may be lost) and writes them and $v \cdot \mathit{end}$ to x (Lines 14-23). In each round of the while loop of Lines $2 - 23$, when a item is read from y (Lines $10-12, 15-20$), it uses *det* to check whether (q_2, W_2) is reached. If so, M_1 return as soon as possible.

The pseudo-code of method M_2 is shown in Method 2. M_2 contains an infinite loop that never returns (Lines 1-3). At each round of the loop, it reads a new update from x and writes it to y.

Method 2: M_2

```
Input: an arbitrary argument
1 while true do
2     $tmp := readOne(x)$;
3     $writeOne(y, tmp)$;
```

9.3 Complexity of k-Bounded TSO-to-TSO linearizability

According to the construction of $\mathcal{L}^M_{(s_1,s_2)}$, we can see that there is a close connection between the paths of lossy simple channel machine and paths of $[\![\mathcal{L}^M_{(s_1,s_2)}, 2]\!]_{tt}$. Fig. 3 shows an example of how to generate a path of $[\![\mathcal{L}^M_{(s_1,s_2)}, 2]\!]_{tt}$ from a path $pa = (q_1, a \cdot b \cdot c) \rightarrow (q_2, b) \rightarrow (q_2, d)$ of simple channel machine M, where $s_1 = (q_1, c, \epsilon, q_3)$ and $s_2 = (q_2, d)$. Here the first transition of *pa* uses rule $rule_1 = (q_1, c, \epsilon, q_3)$ and the second transition of *pa* uses rule $rule_2 = (q_3, \epsilon, d, q_2)$. A path of $[\![\mathcal{L}^M_{(s_1,s_2)}, 2]\!]_{tt}$ is generated as follows:

- M_1 writes $q_1 \cdot start \cdot c \cdot b \cdot a \cdot end$ into x. M_2 reads $q_1 \cdot start \cdot c \cdot b \cdot end$ from x and write them into y. During this process, an item a is lost.
- In the first round of M_1, M_1 reads $q_1 \cdot start \cdot c \cdot b \cdot end$ from y, and according to $rule_1$, M_1 writes $q_3 \cdot start \cdot b \cdot end$ into x. Then M_2 reads $q_3 \cdot start \cdot b \cdot end$ from x and writes them into y.
- In the second round of M_1, M_1 reads $q_3 \cdot start \cdot end$ from y (an item b is lost), and according to $rule_2$, M_1 writes $q_2 \cdot start \cdot d \cdot end$ into x. Since $s_2 = (q_2, d)$ is reached now, M_1 returns.

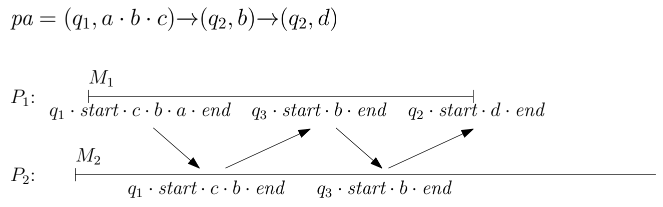

$pa = (q_1, a \cdot b \cdot c) \rightarrow (q_2, b) \rightarrow (q_2, d)$
P_1:
M_1
$q_1 \cdot start \cdot c \cdot b \cdot a \cdot end$
$q_3 \cdot start \cdot b \cdot end$
$q_2 \cdot start \cdot d \cdot end$
P_2:
M_2
$q_1 \cdot start \cdot c \cdot b \cdot end$
$q_3 \cdot start \cdot b \cdot end$

Fig. 3. connection between paths of lossy simple channel machine and paths of $[\![\mathcal{L}^M_{(s_1,s_2)}, 2]\!]_{tt}$

The following lemma states that, there exists an extended history of $[\![\mathcal{L}^M_{(s_1,s_2)}, 2]\!]_{tt}$ that contains a return action, if and only if s_2 is reachable from s_1 in the semantics of lossy simple channel machine M. The proof of this lemma is similar to the proof of Lemma 2 in [13].

Lemma 11. *There exists an extended history eh* $\in$ *ehistory*$([\![\mathcal{L}^{M}_{(s_1,s_2)}, 2]\!]_{tt})$*, such that eh* $\cap \Sigma_{ret} \neq \epsilon$*, if and only if* s_2 *is reachable from* s_1 *in the semantics of lossy simple channel machine* M*.*

Proof. (Sketch) According to the above intuition, given a path pa_m of lossy simple channel machine M, we can generate a path pa_{tt} of $[\![\mathcal{L}^{M}_{(s_1,s_2)}, 2]\!]_{tt}$, such that pa_m is a path from s_1 to s_2, if and only if pa_{tt} contains a return action.

Based on the construction of $\mathcal{L}^{M}_{(s_1,s_2)}$, it is not hard to see that given a path pa_{tt} of $[\![\mathcal{L}^{M}_{(s_1,s_2)}, 2]\!]_{tt}$, we can generate a path pa_m of lossy simple channel machine M. Since M_1 changes "channel contents" according to transition rules of M and returns as soon as reaches s_2, we can see that pa_{tt} contains a return action, if and only if pa_m is a path from s_1 to s_2. This completes the proof of this lemma. $\square$

We construct another concurrent data structure $\mathcal{L}_{pend}$, such that *ehistory*$([\![\mathcal{L}_{pend}, 2]\!]_{tt})$ covers all the extended histories that contains at most two call actions of M_1 and M_2, and never return. $\mathcal{L}_{pend}$ contains two methods M_1 and M_2, and the only sentences in M_1 and M_2 are *while*(*true*);. It is obvious that for any extended history of $\mathcal{L}_{pend}$, there is no return or flushReturn action in it.

We can now prove that k-bounded TSO-to-TSO linearizability has at least non-primitive recursive complexity, as stated by the following theorem:

Theorem 2. *The decision problem of* k*-bound TSO-to-TSO linearizability has at least nonprimitive recursive complexity.*

Proof. (sketch) It is not hard to prove that, *ehistory*$([\![\mathcal{L}_{pend}, 2]\!]_{tt})$ covers all the extended histories that (1) contains at most four actions, while two of them are call actions of M_1 or M_2, and two of them are flushCall actions, (2) on each process, there are at most one call action *call*(i, m, a) and one flushCall action *flushCall*(i', m', a'), and $i = i' \wedge m = m' \wedge a = a'$. Therefore, it is easy to see that, *ehistory*$([\![\mathcal{L}^{M}_{(s_1,s_2)}, 2]\!]_{tt}) \subseteq$ *ehistory*$([\![\mathcal{L}_{pend}, 2]\!]_{tt})$, if and only if no extended history of *ehistory*$([\![\mathcal{L}^{M}_{(s_1,s_2)}, 2]\!]_{tt})$ contains a return action.

Note that if an extended history *eh* of *ehistory*$([\![\mathcal{L}^{M}_{(s_1,s_2)}, 2]\!]_{tt})$ has a return action, then *eh* is not TSO-to-TSO linearizable to any extended history in *ehistory*$([\![\mathcal{L}_{pend}, 2]\!]_{tt})$. Therefore, we can obtain that $\mathcal{L}_{pend}$ 5-bounded TSO-to-TSO linearizes $\mathcal{L}^{M}_{(s_1,s_2)}$ for 2 processes, if and only if no extended history of *ehistory*$([\![\mathcal{L}^{M}_{(s_1,s_2)}, 2]\!]_{tt})$ contains a return action. By Lemma 11, we can see that $\mathcal{L}_{pend}$ 5-bounded TSO-to-TSO linearizes $\mathcal{L}^{M}_{(s_1,s_2)}$ for 2 processes, if and only if s_2 is reachable from s_1 in the semantics of lossy simple channel machine M. This completes the proof of this theorem. $\square$

10 Related Results of other Linearizability on TSO

10.1 k-Bounded TSO-to-SC Linearizability is decidable

Let us propose the notion of well-formed extended histories. 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 flushReturn action *flushReturn*(i_1) matches a flushCall action *flushCall*(i_2), if $i_1 = i_2$. Given an extended history *eh*, let $eh|_i$ be the projection of *eh* to all and only the actions of process i. An extended history *eh* is well-formed, if

- For each i, let $eh_{(c,r,i)} = eh|_i \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})}$. $eh_{(c,r,i)}$ 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.
- For each i, let $eh_{(fc,fr,i)} = eh|_i \uparrow_{(\Sigma_{fcal} \cup \Sigma_{fret})}$. $eh_{(fc,fr,i)}$ starts with a flushCall action and each flushCall (respectively, flushReturn) action is immediately followed by a matching flushReturn (respectively, a flushCall) action unless it is the last action.
- For each i, the number of flushCall actions in $eh|_i$ is less or equal than the number of call action in $eh|_i$, and the number of flushReturn actions in $eh|_i$ is less or equal than the number of call return in $eh|_i$.

A history is well-formed, if it is a well-formed extended history (recall that each history is also an extended history). Our notion of well-formed extended histories is similar to the notion of well-formed histories in [1], and our notion of well-formed histories is same as that in [1]. It is easy to see that, for each concurrent data structure $\mathcal{L}$, each extended history in *ehistory*$([\![\mathcal{L}, n]\!]_{tt})$ is well-formed.

Since each extended history in *ehistory*$([\![\mathcal{L}, n]\!]_{tt})$ is well-formed, given a k-history h, we can see that for each extended history $eh \in ehistory([\![\mathcal{L}, n]\!]_{tt})$, if $eh \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret})} = h$, then *eh* contains at most $2k$ call, return, flushCall and flushReturn actions. According to the proof of Theorem 1, we can obtain *2k-ehistory*$([\![\mathcal{L}, n]\!]_{tt})$. Therefore, we can similarly decide k-bounded TSO-to-SC linearizability, as stated by the following theorem.

Theorem 3. *Given concurrent data structure $\mathcal{L}$ and $\mathcal{L}'$, the problem of whether $\mathcal{L}'$ k-bounded TSO-to-SC linearizes $\mathcal{L}$ for n processes is decidable.*

The following lemma states that k-bounded TSO-to-SC linearizability has nonprimitive recursive complexity.

Theorem 4. *The decision problem of k-bound TSO-to-SC linearizability has at least nonprimitive recursive complexity.*

10.2 Result for Variants of Histories with Bounded Length

Apart from histories and extended histories, other forms of sequences have also been used to represent behaviors of concurrent data structures. For instance, Derrick et. al. [16,17] propose a variant of linearizability on TSO called TSO linearizability. Essentially, TSO linearizability considers a method to start at its call action and end at its flushReturn action. Or we can say, TSO linearizability use sequences of call and flushReturn actions to represent behaviors of concurrent data structures. In this section, we first generalize history and extended history into other forms of sequences, then we prove that we can effectively obtain the set of sequences with bounded length of a concurrent data structure for all above kinds of sequences. At last, we sketch the definition of TSO linearizability, propose a bounded version of TSO linearizability and sketch the proof of its decidability and its at least nonprimitive recursive complexity.

Let us use *cal*, *ret*, *fcal* and *fret* to represent the name of call, return, flush call and flush return actions, respectively. Given distinct $x, y, z, w \in \{cal, ret, fcal, fret\}$, a (x)-history is a sequence of x actions, a (x, y)-history is a sequence of x and y actions, a (x, y, z)-history is a sequence of x, y and z actions, and a (x, y, z, w)-history is a sequence of x, y, z and w actions. For example, a history is a $(call, ret)$-history, while an extended history is a $(call, ret, fcal, fret)$-history. As can be seen, there are in total 15 variants of histories.

A k-(x)-history is a (x)-history that contains at most k actions, and let *k-(x)-history*$(\mathcal{A})$ be the set of k-(x)-histories of $\mathcal{A}$. We can similarly define k-(x, y)-history, k-(x, y, z)-history and k-(x, y, z, w)-history, and we can similarly define *k-(x,y)-history*$(\mathcal{A})$, *k-(x,y,z)-history*$(\mathcal{A})$ and *k-(x,y,z,w)-history*$(\mathcal{A})$. The following lemma states that we can effectively obtain *k-(x)-history*$([\![\mathcal{L}, n]\!]_{tt})$, *k-(x,y)-history*$([\![\mathcal{L}, n]\!]_{tt})$, *k-(x,y,z)-history*$([\![\mathcal{L}, n]\!]_{tt})$ and *k-(x,y,z,w)-history*$([\![\mathcal{L}, n]\!]_{tt})$.

Lemma 12. *Given concurrent data structure $\mathcal{L}$ and distinct $x, y, z, w \in \{cal, ret, fcal, fret\}$, there are algorithms to obtain the sets k-(x)-history$([\![\mathcal{L}, n]\!]_{tt})$, k-(x,y)-history$([\![\mathcal{L}, n]\!]_{tt})$, k-(x,y,z)-history$([\![\mathcal{L}, n]\!]_{tt})$ and k-(x,y,z,w)-history$([\![\mathcal{L}, n]\!]_{tt})$.*

Now let us sketch the definition of TSO linearizability. TSO linearizability relates a set S_1 of "$(cal, fret)$-histories" and a set S_2 of sequential histories. Here we use quotation mark in "$(cal, fret)$-histories" since the flushReturn actions in TSO linearizability is of the form *flushReturn*(i, m, a) and needs to record the method name and the return values. A sequential history is a history 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. Given a "$(cal, fret)$-history" h_1 and a sequential history h_2, and let h_3 be generated from h_1 by transforming each *flushReturn*(i, m, a) action into *return*(i, m, a) for each i, m and a, h_1 is TSO linearizable to h_2, if h_3 is linearizable to h_2. A concurrent data structure $\mathcal{L}$ is TSO linearizable with respect to a set *Spec* of sequential histories for n processes, if for each h_1 of "$(cal, fret)$-histories" of $\mathcal{L}$ for n processes , there exists $h_2 \in Spec$, such that h_1 is TSO linearizable to h_2.

Let us propose the notion of k-bounded TSO linearizability. A concurrent data structure $\mathcal{L}$ is k-bounded TSO linearizable with respect to a set *Spec* of sequential histories (with length less or equal than k) for n processes, if for each h_1 of "k-$(cal, fret)$-histories" of $\mathcal{L}$ for n processes, there exists $h_2 \in Spec$, such that h_1 is 11TSO linearizable" to h_2 (where flushReturn actions are considered as return actions).

Let us sketch why k-bounded TSO linearizability is decidable. Here we require *Spec* to be a regular set. Similarly to the proof of Lemma 12, we can obtain the set S_1 of "k-$(cal, fret)$-histories" of $\mathcal{L}$ for n processes. Since both S_1 and *Spec* are finite sets and the number of their elements are bounded, similarly as Lemma 1, we obtain the decidability result. Similarly as the proof of Theorem 2, we can prove that, $\mathcal{L}_{pend}$ 3-bounded TSO linearizes $\mathcal{L}^M_{(s_1,s_2)}$ for 2 processes, if and only if s_2 is reachable from s_1 in the semantics of lossy simple channel machine M. Therefore, it is easy to see that the decision problem of k-bound TSO linearizability has at least nonprimitive recursive complexity.

11 Conclusion and Future Work

We have shown in this paper that the decision problem of k-bounded TSO-to-TSO linearizability is decidable for bounded number of processes. The proof method is essentially by a reduction to several control state reachability problem of lossy channel machines, which are already known to be decidable. The central idea of our work is to "transform" the two roles of a "composed" call action into a write action and a *cas* action, which belongs to the memory model. During this process we introduce an observer process and bind specific *cas* actions of the observer process with call and return actions. Then, given an extended history, a channel machine M_i^k $(1 \leq i \leq n{+}1)$ is constructed to simulate the k-bounded behaviors of the extended concurrent system from the perspective of each process P_i and check existence of the specific extended history. We then demonstrate that the product of $M_1^{k\text{-}w}, \ldots, M_{n+1}^{k\text{-}w}$, when interpreted with lossy channels, can characterize the existence of the extended history from a specific begin state to a specific end state (both with empty buffer for each process). Therefore, the existence of a specific extended history can be obtained by repeatedly applying above process with enumerating all possible start and end states, and the set of k-extended histories can be obtained by enumerating all k-extended histories and use check their existence one by one. The decidability result of k-bounded TSO-to-TSO linearizability follows from that we can obtain the set of k-extended histories of a concurrent data structure.

Since existence of a k-history is equivalent to existence of a $2k$-extended history with same history, we can obtain the set of k-histories of a concurrent data structure and then decide k-bounded TSO-to-SC linearizability. Similarly we prove that k-bounded TSO linearizability is also decidable. We prove that all above three bounded variants of linearizability on TSO has at least nonprimitive recursive complexity by a reduction from a reachability problem of a lossy simple channel machine, which is known to have nonprimitive recursive complexity. Since these bounded variants do not require the size of a store buffer or the length of a trace of a concurrent system to be bounded, it still allows infinite-state behaviors. Hence, our decidability result is non-trivial. As by-product of our work, we prove that we can effectively obtain the set of sequences with bounded length of a concurrent data structure for all fifteen variants of histories. This sheds light on developing algorithms for automatically verifying concurrent libraries on relaxed memory models for (bounded) variants of linearizabilities.

References

1. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12** (1990) 463–492
2. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programm. IEEE Transactions on Computers **28** (1979) 690–691
3. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: TPHOLs. (2009) 391–407
4. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding power multiprocessors. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11, New York, NY, USA, ACM (2011) 175–186

5. Batty, M., Dodds, M., Gotsman, A.: Library abstraction for c/c++ concurrency. In: POPL. (2013) 235–248
6. Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '05, New York, NY, USA, ACM (2005) 378–391
7. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent library correctness on the tso memory model. In: ESOP. (2012) 87–107
8. Gotsman, A., Musuvathi, M., Yang, H.: Show no weakness: Sequentially consistent specifications of TSO libraries. In: Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings. (2012) 31–45
9. Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. In: ESOP. (2009) 252–266
10. Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science. LICS '96, Washington, DC, USA, IEEE Computer Society (1996) 219–
11. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13, Berlin, Heidelberg, Springer-Verlag (2013) 290–309
12. Wang, C., Lv, Y., Wu, P.: Bounded tso-to-sc linearizability is decidable. Technical Report ISCAS-SKLCS-15-11, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences (2015)
13. Wang, C., Lv, Y., Wu, P.: Tso-to-tso linearizability is undecidable. In: Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Proceedings. (2015) 264–280
14. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '10, New York, NY, USA, ACM (2010) 7–18
15. Schnoebelen, P.: Verifying lossy channel systems has nonprimitive recursive complexity. Inf. Process. Lett. **83** (2002) 251–261
16. Derrick, J., Smith, G., Groves, L., Dongol, B.: Using coarse-grained abstractions to verify linearizability on TSO architectures. In Yahav, E., ed.: Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings. Volume 8855 of Lecture Notes in Computer Science., Springer (2014) 1–16
17. Derrick, J., Smith, G., Dongol, B.: Verifying linearizability on TSO architectures. In Albert, E., Sekerinski, E., eds.: Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings. Volume 8739 of Lecture Notes in Computer Science., Springer (2014) 341–356
18. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. In Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B., eds.: Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II. (2015) 95–107
19. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. (2015) 651–662
20. Atig, M.F.: What is decidable under the TSO memory model? ACM SIGLOG News **7** (2020) 4–19

A Definitions and Proofs in Section 8

A.1 Construction of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_g$

It is quite hard to build a weak simulation relation or something alike between states of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$ and configurations of $M_1^{\text{k-(f,exth)}} \otimes M_{n+1}^{\text{k-(f,exth)}}$. This is because that for a state of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$, more than one process may be possible to do a flush action at any time. Therefore, the content of buffers can be flushed out in many orders, and the total store orders of traces from this state is not fixed. While for a state of $M_1^{\text{k-(f,exth)}} \otimes M_{n+1}^{\text{k-(f,exth)}}$, since the flush actions are launched according to a single channel, items must be flush in a fixed FIFO order, and the total store orders of traces from this state is to a large extent fixed.

To deal with this problem, a intermediate transition system is introduced, whose configuration extends configurations of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$ and contains the total store order of some trace. Formally, given a concurrent data structure $\mathcal{L} = (\mathcal{X}_\mathcal{L}, \mathcal{M}, \mathcal{D}_\mathcal{L}, Q_\mathcal{L}, \rightarrow_\mathcal{L}$), and a deterministic finite state automaton $\mathcal{A}_{eh} = (Q_s, \Sigma_s, \rightarrow_s, F_s, q_{is})$ that is constructed for extended history *eh*, as in Subsection 7.2. We construct an LTS $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_g = (\mathit{Conf}_g, \Sigma_g, \rightarrow_g, \mathit{InitConf}_g)$. Here $\Sigma_g = \Sigma_f$, $\mathit{InitConf}_g = (p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon)$. Conf_g and $\rightarrow_g$ are defined as follows.
Construction of Conf_g: Each configuration of Conf_g is a tuple $(p, d, u, q_s, \mathit{mak}, \mathit{flag}, g)$, where,

- (p, d, u, mak) is a configuration of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$, Here $\mathit{mak} \in \mathit{markedVal}(\mathcal{M}, \mathcal{D}_\mathcal{L}, n)$ is used to ensure that each specific $\mathit{cas}(n{+}1, z_w, \_)$ action is immediately followed by the corresponding call or return action.
- $q_s \in Q_s$ is used to monitor whether the extended history *eh* happens,
- $g \in (\Sigma_{e0} \cup \Sigma_{e1} \cup \Sigma_{e2} \cup \Sigma_{e3})^*$, where Σ_{e0}, Σ_{e1}, Σ_{e2} and Σ_{e3} are defined below. g is the total store order of some execution, and it should satisfies some requirements shown below.
- $\mathit{flag} \in \{T, F\}$ is used to denote whether the total store order g has been initialized.

The four alphabets of $\Sigma_{e0}, \Sigma_{e1}, \Sigma_{e2}, \Sigma_{e3}$ is defined as follows:

- $\Sigma_{e0} = \{(i, x, d) | 1 \leq i \leq n{+}1, x \in \mathcal{X}_\mathcal{L} \cup \{z_w, z_f\}, d \in \mathit{Val}\}$ represents the items in total store order of a trace that are not used now and will be flushed later than any item in current buffer.
- $\Sigma_{e1} = \{(i, x, d)' | (i, x, d) \in \Sigma_{e0}\}$ represents items in the total store order of a trace that are not used now and will be flushed earlier than some item in buffer.
- $\Sigma_{e2} = \{(i, x, d)'' | (i, x, d) \in \Sigma_{e0}\}$ represents items in the total store order of a trace that are already inserted into buffer and not flushed yet.
- $\Sigma_{e3} = \{(i, x, d)''' | (i, x, d) \in \Sigma_{e0}\}$ represents items in the total store order of a trace that have already been flushed out from buffer.

Requirements of g: g stores the total store order of a trace. It is a concatenation of sequences l_{g1}, l_{g2} and l_{g3}. $l_{g1} \in \Sigma_{e0}^*$ represents the sequence of items that have not been used and will be flushed later than any item in current buffer. $l_{g2} \in (\Sigma_{e1} \cup \Sigma_{e2})^*$ represents the sequence of items that either in current buffer, or items not in current buffer but will be flushed earlier than some item in current buffer. $l_{g3} \in \Sigma_{e3}^*$ represents the sequence of items that have already been flushed.

Given a finite sequence $l = \alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_k$, we say that the element α_i is left (resp., right) to element α_j, if $i < j$ (resp., $i > j$). We say that α_i is left most element in l if $i = 1$, and α_i is right most element in l if $i = |l|$.

Let Σ_i be the items of process i in $\Sigma_{e0} \cup \Sigma_{e1} \cup \Sigma_{e2} \cup \Sigma_{e3}$, $\Sigma_{(i,x)}$ be the items of process i and memory location x in $\Sigma_{e0} \cup \Sigma_{e1} \cup \Sigma_{e2} \cup \Sigma_{e3}$, g should additionally satisfy the following requirements (we briefly explain each requirement):

- If $l_{g2} \neq \epsilon$, then $l_{g2}(1) \in \Sigma_{e2}$. To explain this, assume that if $l_{g2}(1) = a \in \Sigma_{e1}$, then a will be flushed later than any item in buffer, and this violates our intuition of Σ_{e1}.
- For each i, $l_{g2} \uparrow_{\Sigma_i} \in \Sigma_{e1}^* \cdot \Sigma_{e2}^*$. This is obvious from our intuition of Σ_{e1} and Σ_{e2}.
- For each i, j, if $g \uparrow_{(\Sigma_i \cup \Sigma_{e2})} (j) = (i, x, d)''$ and $d(x) = a$, then $u(i)(j) = (i, x, a)$, and vice versa. This is obvious from our intuition of Σ_{e2}.
- Let g' be the sequence generated from g by discarding all the $'$ symbols of each item in g. If $g'(|g'|) = (i_1, x_1, d_1)$, then $d_1 = d_{init}[x_1 : d_1(x_1)]$. For each i, if $g'(i) = (i_2, x_2, d_2)$, $g'(i{+}1) = (i_3, x_3, d_3)$, then $d_2 = d_3[x_2 : d_2(x_2)]$. Or we can say, the valuation tuple of g_i is generated from d_{init} by applying updates in $g(|g'|), \ldots, g(i{+}1)$.

Construction of $\rightarrow_g$: Recall that $\rightarrow_b$ is the transition relation in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$. The transition relation $\rightarrow_g$ is defined as follows:

- Initial transition: the first transition from $InitConf_g$ is to guess the tuple of total store order: $(p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon) \xrightarrow{\epsilon}_g (p_{init}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, T, g)$, where g is a sequence that satisfy the requirements in previous several paragraphs. The *flag* tuple ensure that this transition can be carried out only once.
- τ and read transitions: $(p, d, u, q_s, \epsilon, T, g) \xrightarrow{act}_g (p', d, u, q_s, \epsilon, T, g)$, and *act* is a τ or read action, if $(p, d, u, \epsilon) \xrightarrow{act}_b (p', d, u, \epsilon)$.
- Write transitions: $(p, d, u, q_s, \epsilon, T, g) \xrightarrow{write(i,x,a)}_g (p', d, u', q_s, \epsilon, T, g')$, if $(p, d, u, \epsilon) \xrightarrow{write(i,x,a)}_b (p', d, u', \epsilon)$, and one of the following conditions holds: (1) $l_{g2} \uparrow_{\Sigma_{(i,x)}}$ contains at least one item in Σ_{e1}, the right most item of $l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_{(i,x)})} = (i, x, d_1)'$ and $d_1(x) = a$, and g' is generated from g by transforming this item into $(i, x, d_1)''$, (2) $l_{g2} \uparrow \Sigma_{(i,x)}$ does not contain any item of Σ_{e1}, the right most item of $l_{g1} \uparrow_{(\Sigma_{e0} \cap \Sigma_{(i,x)})} =$ (i, x, d_2) with $d_2(x) = a$, g' is generated from g by transforming this item into $(i, x, d_1)''$, and mark all the items which are right to this item in l_{g1} with $'$ symbol. Write transition will put a item *itm* into buffer, or we can say, mark *itm* with $''$ symbol. In this first case, *itm* is already marked with $'$ symbol before write transition, and we only need to mark *itm* with $''$ symbol. In the second case, *itm* belongs to l_{g1} before write transition, and we mark *itm* with $''$ symbol; the items which are right to *itm* in l_{g1} means the items that should be flushed earlier than *itm*, so we mark them with $'$ symbol.
- *cas* transitions: $(p, d, u, q_s, mak, T, g) \xrightarrow{cas(i,x,a,b)}_g (p', d', u, q_s, mak', T, g')$, if (p, d, u, mak) $\xrightarrow{cas(i,x,a,b)}_b (p', d', u, mak)$, and one of the following conditions holds: (1) l_{g2} ends

with $(i, x, d')'$, $d'(x) = b$, and g' is generated from g by transforming this item into $(i, x, d')'''$, (2) $l_{g2} = \epsilon$, l_{g1} ends with (i, x, d'), $d'(x) = b$, and g' is generated from g by transforming this item into $(i, x, d')'''$.
Note that, according to $\rightarrow_b$, $x \neq z_f$. If $x \neq z_w$, then $mak = mak' = \epsilon$. Otherwise, $mak = \epsilon$ and $mak' \in markedVal(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n)$.
cas transition will flush a item *itm* out from buffer, or we can say, mark *itm* with $'''$ symbol. In the first case, *itm* is chosen from l_{g2}. In the second case, *itm* is chosen from l_{g1}.

- Flush transitions: $(p, d, u, q_s, \epsilon, T, g) \xrightarrow{flush(i,x,a)}_g (p, d', u', q_s, \epsilon, T, g')$, if $x \neq z_f$, $x \neq z_w$, l_{g2} ends with $(i, x, d')''$, $d'(x) = a$, and g' is generated from g by transforming this item into $(i, x, d')'''$.
  Flush transition will flush a item *itm*, which is already in buffer (marked with $''$ symbol), out from buffer, or we can say, mark *itm* with $'''$ symbol. When *itm* is a item of z_w, we must modify the *mak* tuple to make sure that the next transition be a corresponding call or return transition.
- FlushCall and flushReturn transitions: $(p, d, u, q_s, \epsilon, T, g) \xrightarrow{flushCall(i)}_g (p, d', u', q'_s, \epsilon, T, g')$, if $(p, d, u, \epsilon) \xrightarrow{flushCall(i)}_b (p, d', u', \epsilon)$, $q_s \xrightarrow{flushCall(i)}_s q'_s$, l_{g2} ends with $(i, z_f, d')''$, $d'(z_f) = call$, and g' is generated from g by transforming this item into $(i, z_f, d')'''$.
  Similarly, $(p, d, u, q_s, \epsilon, T, g) \xrightarrow{flushReturn(i)}_g (p, d', u', q'_s, \epsilon, T, g')$, if $(p, d, u, \epsilon) \xrightarrow{flushReturn(i)}_b (p, d', u', \epsilon)$, $q_s \xrightarrow{flushReturn(i)}_s q'_s$, l_{g2} ends with $(i, z_f, d')''$, $d'(x) = ret$, and g' is generated from g by transforming this item into $(i, z_f, d')'''$.
  FlushCall or FlushReturn transition will flush a item *itm* of z_f with z_f in valuation of *item* being *call* or *ret*, respectively. *itm* must be already in buffer (marked with $''$ symbol) before transition, and will be flushed out (marked with $'''$ symbol) after transition.
- Call and return transitions: $(p, d, u, q_s, call(i, m, a), T, g) \xrightarrow{call(i,m,a)}_g (p', d', u', q'_s, \epsilon, T, g)$, if $(p, d, u, call(i, m, a)) \xrightarrow{call(i,m,a)}_b (p', d', u', \epsilon)$ and $q_s \xrightarrow{call(i,m,a)}_s q'_s$. Similarly, $(p, d, u, q_s, return(i, m, a), T, g) \xrightarrow{return(i,m,a)}_g (p', d', u', q'_s, \epsilon, T, g)$, if $(p, d, u, return(i, m, a)) \xrightarrow{return(i,m,a)}_b (p', d', u', \epsilon)$ and $q_s \xrightarrow{return(i,m,a)}_s q'_s$.
  A call or return action can happen, if the value of *mak* tuple before transition is the corresponding call or return action, and we unset *mak* tuple into ϵ after transition. This ensure that a call or return action is “bound” with a corresponding *cas* action of z_w(which set the *mak* tuple into the value of the corresponding call or return action).

A.2 Proof of Lemma 5

Given a sequence g that satisfies requirement in Appendix A.1 and a sequence t of actions, we say that g is the total store order of t, if the the following conditions are satisfied:

- Let t' be the projection of t into write and *cas* actions. g contains $|t'|$ elements, and let g' be generated from g by transforming each element (a, b, c), $(a, b, c)'$, $(a, b, c)''$ or $(a, b, c)'''$ in g into (a, b, c).

- If $t'(1) = \mathit{write}(j,x,b)$ or $\mathit{cas}(j,x,a,b)$, then $g'(|t'|) = (j,x,d)$, where $d = d_{\mathit{init}}[x:b]$.
- For each $i > 1$, if $t'(i) = \mathit{write}(j,x,b)$ or $\mathit{cas}(j,x,a,b)$, and $g(|t'|-i+2) = (j',y,d')'''$, then $g'(|t'|-i+1) = (j,x,d)$, where $d = d'[x:b]$.

The following lemma states that if t is a marked witness of an extended history *eh* in $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$, then t is also a trace of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_g$.

Lemma 13. *Given a k-extended history eh. If t is a marked witness of eh from* $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ *to* $(p_w, d_w, \epsilon^{n+1}, \epsilon)$ *in* $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$*, then t is also a trace from* $(p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon)$ *to* $(p_w, d_w, \epsilon^{n+1}, q_{ends}, \epsilon, T, g)$ *in* $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_g$*, where g is the total store order of t.*

Proof. To prove this lemma, for each path of t in $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$ from $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, \epsilon)$, we generate a path of $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_g$ from $(p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, q_{ends}, \epsilon, T, g)$ step by step. Note that each element in g is of the form $(\_,\_,\_)$.

Assume $(p_{init}, d_{init}, \epsilon^{n+1}, \epsilon) \xrightarrow{act_1}_b (p_1, d_1, u_1, \mathit{mak}_1) \ldots \xrightarrow{act_w}_b (p_w, d_w, u_w, \mathit{mak}_w)$ is the path of t in $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_b$. Here $u_w = \epsilon^{n+1}$. For each configuration $(p_i, d_i, u_i, \mathit{mak}_i)$, we construct another configuration $(p_i, d_i, u_i, q_s^i, \mathit{mak}_i, T, g_i)$, where

- q_s^i is generated from q_{is} by call, return, flushCall and flushReturn actions in $\mathit{act}_1 \cdot \ldots \cdot \mathit{act}_i$.
- If $\mathit{act}_i = \mathit{cas}(n{+}1, z_w, \_, \alpha)$ and $\alpha \in \mathit{markedVal}(\mathcal{M}, \mathcal{D}_{\mathcal{L}}, n)$, then $\mathit{mak}_i = \alpha$. Otherwise, $\mathit{mak}_i = \epsilon$.
- Let us show how to construct g_i from g.
  - l_{gi3} contains all the items that has been flushed when reaching $(p_i, d_i, u_i, \mathit{mak}_i)$.
  - To construct l_{gi2}, we use $''$ symbol to mark all the item that has been putted into buffer. Let ind_1 be the minimal index that has been marked with $''$, and let ind_2 be the minimal index that has been marked with $'''$. For all the items that (1) has index larger than ind_1 and bigger than ind_2 and (2) has not been marked with $''$, we mark them with $'$. l_{gi2} starts from ind_1 and ends with ind_2-1.
  - The remaining part is l_{gi1}.

It is obvious that $(p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon) \xrightarrow{\epsilon}_g (p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, T, g_0)$. Here g_0 is the total store order of t and every element in g_0 is of the form $(\_,\_,\_)$. It is also easy to see that for each i, $(p_i, d_i, u_i, q_s^i, \mathit{mak}_i, T, g_i) \xrightarrow{act_{i+1}}_e (p_{i+1}, d_{i+1}, u_{i+1}, q_s^{i+1}, \mathit{mak}_{i+1}, T, g_{i+1})$. Therefore, $t = \mathit{act}_1 \cdot \ldots \cdot \mathit{act}_w$ is a trace from $(p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, q_{ends}, \epsilon, T, g)$ in $[\![\mathit{Clt}_f(\mathit{Mod}(\mathcal{L})), n{+}1]\!]_g$. □

Given a sequence g as defined in Appendix A.1 and a sequence l of flush, call, return, flushCall and flushReturn actions in $M_i^{k\text{-}(f,exth)}$, we say that l is consistent with g, if:

- Let g' be a sequence generated from g by discarding $'$ symbol of each item in g. Let l_f be a projection of l into flush, flushCall and flushReturn actions.

- Construct sequences of valuations $val_0, val_1, \ldots$. $val_0 = d_{init}$. For each i, if $l_f(i) = \mathit{flush}(i,x,d)$, then $Val_i = d$; else, if $l_f = \mathit{flushCall}(i)$, then $Val_i = Val_{i\text{-}1}[z_f : cal]$; otherwise, we can see $l_f = \mathit{flushReturn}(i)$, and then $Val_i = Val_{i\text{-}1}[z_f : ret]$.
- Let g' be a sequence generated from g by discarding $'$ symbol of each item in g. Then for each i, if $l_f(i) = \mathit{flush}(i,x,d)$, then $g'(|g'|-i+1) = (i,x,Val_i)$; else, if $l_f = \mathit{flushCall}(i)$, then $g'(|g'|-i+1) = (i,z_f,Val_i)$; else, if $l_f = \mathit{flushReturn}(i)$, then $g'(|g'|-i+1) = (i,z_f,Val_i)$;

Or we can say, l is consistent with g, if the flush, flushCall and flushReturn transitions in l can be used to generate g.

The following lemma states that, given a trace t of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$, there exists a sequence $l \in T^{(S,K_1)}_{(q_i,q'_i)} M_i^{k\text{-}(f,exth)}$, such that l is consistent with g, which is the total store order of t.

Lemma 14. *Given a k-extended history eh. Assume that t is a trace from $(p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, q_{ends}, \epsilon, T, g)$ in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$, and g is the total store order of t, and assume that t is a marked witness of eh. Then there exists a sequence l, such that for each process id $1 \le i \le n{+}1$, $l \in T^{(S,K_1)}_{(q_i,q'_i)} M_i^{k\text{-}(f,exth)}$, where $q_i = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon)$, $q'_i = (p_w(i), d_w, d_w, q_{ends}, \epsilon)$, and l is consistent with g.*

Proof. This lemma is proved by constructing a weak simulation between configurations of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$ in the path of t and configurations of (S,K_1)-channel machine $M_1^{k\text{-}(f,exth)} \otimes \ldots \otimes M_{n+1}^{k\text{-}(f,exth)}$.

Assume $(p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon) \xrightarrow{act_1}_g \ldots \xrightarrow{act_w}_g (p_w, d_w, u_w, q_{ends}, \epsilon, T, g)$ is the path of t in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$ and $u_w = \epsilon^{n+1}$. Let $(p, d, u, r, q_s, mak, T, g)$ be the *(v+1)-th* configuration of the path. Let $((cs_1, \ldots, cs_{n+1}), (c_1, \ldots, c_{n+1}))$ be a configuration of (S,K_1)-channel machine $M_1^{k\text{-}(f,exth)} \otimes \ldots \otimes M_{n+1}^{k\text{-}(f,exth)}$, and for each i, $cs_i = (q_i, d_{ci}, d_{gi}, q_s^i, mak_i, cnt_i)$. A relation $\sim$ is defined as follows: $(p, d, u, q_s, mak, T, g) \sim ((cs_1, \ldots, cs_{n+1}), (c_1, \ldots, c_{n+1}))$, if for each process id i,

- $p(i) = q_i$, $d = d_{ci}$, $q_s = q_s^i$, $mak = mak_i$, and d_{gi} is generated from d_{ci} by doing all the updates in $c_i(|c_i|), \ldots, c_i(1)$.
- $cnt_1 = \ldots = cnt_{n+1}$. And cnt_1 is the number of call, return, flushCall and flushReturn actions in $act_1 \cdot \ldots \cdot act_v$.
- If $l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)} \neq \epsilon$, then let ind_1, ind_2 be the index of the leftmost $\Sigma_{e2} \cap \Sigma_i$ item on g and the rightmost item on l_{g2}, respectively, let g' be generated from g by discarding $'$ symbols of each item in g. Assume $g' = (i_1, x_1, d_1) \cdot (i_2, x_2, d_2) \cdot \ldots$. Then c_i contains $ind_2 - ind_1 + 1$ items, and $\forall 1 \le j \le ind_2 - ind_1 + 1$, $c_i(ind_2 - ind_1 - j + 2) = (i_{(ind_2-j+1)}, x_{(ind_2-j+1)}, d_{(ind_2-j+1)})$ or $((i_{(ind_2-j+1)}, x_{(ind_2-j+1)}, d_{(ind_2-j+1)}), \sharp)$.
- If $l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)} = \epsilon$ and $l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_i)} \neq \epsilon$, then let ind_1, ind_2 be the index of the rightmost $\Sigma_{e1} \cap \Sigma_i$ item on g and the rightmost item on l_{g2} respectively, let g' be generated from g by discarding $'$ symbols of each item in g. Assume $g' = (i_1, x_1, d_1) \cdot (i_2, x_2, d_2) \cdot \ldots$. Then c_i contains $ind_2 - ind_1$ items, and $\forall 1 \le j \le ind_2 - ind_1$, $c_i(ind_2 - ind_1 - j + 1) = (i_{(ind_2-j+1)}, x_{(ind_2-j+1)}, d_{(ind_2-j+1)})$ or $((i_{(ind_2-j+1)}, x_{(ind_2-j+1)}, d_{(ind_2-j+1)}), \sharp)$.

- If $l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)} = \epsilon$ and $l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_i)} = \epsilon$, then let ind_1, ind_2 be the index of the leftmost item on l_{g2} and the rightmost item on l_{g2} respectively, let g' be generated from g by discarding $'$ symbols of each item in g. Assume $g' = (i_1, x_1, d_1) \cdot (i_2, x_2, d_2) \cdot \ldots$. Then c_i contains $ind_2 - ind_1 + 1$ items, and $\forall 1 \leq j \leq ind_2 - ind_1 + 1$, $c_i(ind_2 - ind_1 - j + 2) = (i_{(ind_2-j+1)}, x_{(ind_2-j+1)}, d_{(ind_2-j+1)})$ or $((i_{(ind_2-j+1)}, x_{(ind_2-j+1)}, d_{(ind_2-j+1)}), \sharp)$.

It remains to prove that, if $(p, d, u, r, q_s, mak, T, g) \sim ((cs_1, \ldots, cs_{n+1}), (c_1, \ldots, c_{n+1}))$ holds, $(p, d, u, r, q_s, mak, T, g) \xrightarrow{act_{v+1}}_g (p', d', u', r', q'_s, mak', T, g')$ and $(p', d', u', r', q'_s, mak', T, g')$ is the *v+2-th* configuration of the path of t, then there exists $cs'_1, \ldots, cs'_{n+1}, c'_1, \ldots, c'_{n+1}$ and a sequence s_{v+1}, such that $((cs_1, \ldots, cs_{n+1}), (c_1, \ldots, c_{n+1})) \xrightarrow{s_{v+1}}{}^*_{\Delta^k_i} (cs'_1, \ldots, cs'_{n+1}), (c'_1, \ldots, c'_{n+1})), (p', d', u', r', q'_s, mak', T, g') \sim ((cs'_1, \ldots, cs'_{n+1}), (c'_1, \ldots, c'_{n+1}))$ holds, and the flush, call, return, flushCall and flushReturn actions in act_{v+1} is same to that in s_{v+1}. Assume for each i, $cs'_i = (q'_i, d'_{ci}, d'_{gi}, q^{i'}_s, mak'_i, cnt'_i)$.

- When α_{v+1} is a τ, *read*, call or return action, it is obvious to see that $s_{v+1} = \epsilon$ and this holds trivially.
- When α_{v+1} is a write actions of process i, $s_{v+1} = \epsilon$, and $M^{k\text{-}(f,exth)}_1 \otimes \ldots \otimes M^{k\text{-}(f,exth)}_{n+1}$ do the following transitions and change the channels as follows:
  - If $l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_i)} \neq \epsilon$, then let ind_1 be the index of the right most $\Sigma_{e1} \cap \Sigma_i$ item on g, let ind_2 be
    - ind'-1, if $l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)} \neq \epsilon$ and ind' is the index of the leftmost $\Sigma_{e2} \cap \Sigma_i$ item on g,
    - ind_1, if $l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)} = \epsilon \wedge l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_i)} \neq \epsilon$,

    c'_i is generated from c_i by putting updates of $g(ind_2), \ldots, g(ind_1)$ into c_i. During this process,several guess write actions and then a write action happen (seen as internal transitions in $M^{k\text{-}(f,exth)}_1 \otimes \ldots \otimes M^{k\text{-}(f,exth)}_{n+1}$). For channel $j \neq i$, $c'_j = c_j$.
  - If $l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_i)} = \epsilon$,
    - For channel i: Let ind_1 be the index of the right most Σ_i item of l_{g1} in g, let ind_2 be
      - ind'-1, if $l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)} \neq \epsilon$ and ind' is the index of the leftmost $\Sigma_{e2} \cap \Sigma_i$ item on g,
      - ind'-1, if $l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)} = \epsilon$ and ind' is the index of the leftmost item of l_{g2} on g,

      c'_i is generated from c_i by putting updates of $g(ind_2), \ldots, g(ind_1)$ into c_i. During this process, several guess write actions and then a write actions happen (seen as internal transitions in $M^{k\text{-}(f,exth)}_1 \otimes \ldots \otimes M^{k\text{-}(f,exth)}_{n+1}$).
    - For channel $j \neq i$:
      - If $l_{g2} \uparrow_{\Sigma_j} = \epsilon$. Let ind_1 be the index of the right most Σ_i item of l_{g1} in g. Let ind_2 be the index of right most item of l_{g1} in g. Let sequence $g'' = g(ind_1) \cdot \ldots \cdot g(ind_2)$.
        If $g'' \uparrow_{\Sigma_j} \neq \epsilon$, let ind_3 be the index of right most item of $g'' \uparrow_{\Sigma_j}$ in g, and c'_j is generated from c_j by putting updates of $g(ind_2), \ldots, g(ind_3{+}1)$. During this process several guess write actions happen (seen as internal transitions in $M^{k\text{-}(f,exth)}_1 \otimes \ldots \otimes M^{k\text{-}(f,exth)}_{n+1}$).

Otherwise, if $g'' \uparrow_{\Sigma_j} = \epsilon$, c'_j is generated from c_j by putting updates of $g(\mathit{ind}_2), \ldots, g(\mathit{ind}_1)$. During this process several guess write actions happen (seen as internal transitions in $M_1^{\mathit{k\text{-}(f,exth)}} \otimes \ldots \otimes M_{n+1}^{\mathit{k\text{-}(f,exth)}}$).

- For channel $j \neq i$ and $l_{g2} \uparrow_{\Sigma_j} \neq \epsilon$ holds: $c'_j = c_j$.

- When act_{v+1} is a flush action of process i, then $s_{v+1} = \mathit{act}_{v+1}$. $M_1^{\mathit{k\text{-}(f,exth)}} \otimes \ldots \otimes M_{n+1}^{\mathit{k\text{-}(f,exth)}}$ do the following transitions and change the channels as follows:
  - For channel $j \neq i$, c'_j is generated from c_j by discarding the right most item of c_j. During this process a flush action happens.
  - For channel i, c'_i is generated from c_i as follows:
    - If $|l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)}| \geq 2$, then c'_i is generated from c_i by discarding the right most item. During this process a flush action happens.
    - Otherwise, $|l_{g2} \uparrow_{(\Sigma_{e2} \cap \Sigma_i)}| = 1$.
      Let ind_1 be the index of the right most $\Sigma_{e1} \cap \Sigma_i$ item in g if $l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_i)} \neq \epsilon$. Otherwise, let ind_1 be the index of the right most item of l_{g1} in g. Let ind_2 be the index of the $\Sigma_{e2} \cap \Sigma_i$ item in g. c'_i is generated from c_i be putting updates of $g(\mathit{ind}_2 - 1), \ldots, g(\mathit{ind} + 1)$ and discarding the right most item of c_i. During this process several guess write actions (seen as internal transitions in $M_1^{\mathit{k\text{-}(f,exth)}} \otimes \ldots \otimes M_{n+1}^{\mathit{k\text{-}(f,exth)}}$) and a flush action happen.
- When act_{v+1} is a flushCall or flushReturn action of process i, since such actions come from flushing z_f, the change to channel is similar to the case of flush actions, and we launch flushCall or flushReturn transition.
- When act_{v+1} is a $\mathit{cas}(i, x, \mathit{val})$ action of process i, then $s_{v+1} = \mathit{flush}(i, x, \mathit{val})$. $M_1^{\mathit{k\text{-}(f,exth)}} \otimes \ldots \otimes M_{n+1}^{\mathit{k\text{-}(f,exth)}}$ do the following transitions and change the channels as follows:
  - If $l_{g2} = \epsilon$. For channel $j \neq i$, $c'_j = c_j = \epsilon$, and during transition the update (i, x, val) need to be inserted into c'_j by a guess write action (seen as internal transitions in $M_1^{\mathit{k\text{-}(f,exth)}} \otimes \ldots \otimes M_{n+1}^{\mathit{k\text{-}(f,exth)}}$) and then flushed our of c'_j using a flush action. For channel i, $c'_i = c_i = \epsilon$, and during this process a flush action happens.
  - If $l_{g2} \neq \epsilon$, then it is easy to see that l_{g2} ends with a item in $\Sigma_{e1} \cap \Sigma_i$. For channel $j \neq i$, c'_j is generated from c_j by discarding the right most item using a flush action. For channel i, the channel c'_i is generated as follows:
    - If $|l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_i)}| \geq 2$, let ind_1 be the index of the second right most $\Sigma_{e1} \cap \Sigma_i$ item in g, let ind_2 be the index of the right most $\Sigma_{e1} \cap \Sigma_i$ item in g. c'_i is generated from c_i by putting the updates of $g(\mathit{ind}_2 - 1), \ldots, g(\mathit{ind} + 1)$ into c_i. During this process several guess write action happen (seen as internal transitions in $M_1^{\mathit{k\text{-}(f,exth)}} \otimes \ldots \otimes M_{n+1}^{\mathit{k\text{-}(f,exth)}}$) and then a flush action happends.
    - If $l_{g2} \uparrow_{(\Sigma_{e1} \cap \Sigma_i)} = \epsilon$, let ind_1 be the index of the right most item of l_{g1} in g, let ind_2 be the index of the $\Sigma_{e1} \cap \Sigma_i$ item in g. c'_i is generated from c_i by putting the updates of $g(\mathit{ind}_2 - 1), \ldots, g(\mathit{ind} + 1)$ into c_i. During this process several guess write actions happen (seen as internal transitions in $M_1^{\mathit{k\text{-}(f,exth)}} \otimes \ldots \otimes M_{n+1}^{\mathit{k\text{-}(f,exth)}}$) and then a flush action happens.

When act_{v+1} is a cas action of process i for z_f, the mak tuple of each $M_i^{\mathit{k\text{-}(f,exth)}}$ must also be changed. Since flush action requires the mak tuples to be ϵ, in the case when $l_{g2} \neq \epsilon$ and for process i, we do guess write transitions first and then flush transition.

According to our construction, given a path *pa* of t from $(p_{in}, d_{init}, \epsilon^{n+1}, q_{is}, \epsilon, F, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, q_{ends}, \epsilon, T, g)$ in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_g$, we can construct a path pa' of $M_1^{k\text{-}(f,exth)} \otimes \ldots \otimes M_{n+1}^{k\text{-}(f,exth)}$, and the call, return, flushCall and flushReturn actions in *pa* equals that in pa'. Let t' be the trace of pa'. It is obvious that $t' \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})} = eh$. Therefore, the q_s tuple in the end state of pa' is q_{ends} for each process. Since *eh* contains less or equal than k actions, the number of call, return, flushCall and flushReturn actions in t' is less or equals than k. Therefore, channels along states of pa' satisfies strong symbol restriction (S, K_1). This completes the proof of this lemma. $\square$

With above two lemmas we can now prove Lemma 5.

Lemma 5. *Given a k-extended history eh. If there exists a marked witness t of eh from* $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ *to* $(p_w, d_w, \epsilon^{n+1}, \epsilon)$ *in* $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, *then* $\cap_{i=1}^{n+1} T_{(q_i, q'_i)}^{(S,K_1)} M_i^{k\text{-}(f,exth)} \neq \emptyset$, *where for each* $1 \leq i \leq n{+}1$, $q_i = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon, 0)$, $q'_i = (p_w(i), d_w, d_w, q_{ends}, \epsilon, |t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})}|)$.

Proof. Lemma 5 is a direct consequence of Lemma 13 and Lemma 14. $\square$

A.3 Proof of Lemma 6

Lemma 6. *Given a k-extended history eh. If* $\cap_{i=1}^{n+1} T_{(q_i, q'_i)}^{(S,K_1)} M_i^{k\text{-}(f,exth)} \neq \emptyset$, *where for each* $1 \leq i \leq n{+}1$, $q_i = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon, 0)$, $q'_i = (p_w(i), d_w, d_w, q_{ends}, \epsilon, |eh|)$, *then there exists a marked witness t of eh from* $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ *to* $(p_w, d_w, \epsilon^{n+1}, \epsilon)$ *in* $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$.

Proof. Since $\cap_{i=1}^{n+1} T_{(q_i, q'_i)}^{(S,K_1)} M_i^{k\text{-}(f,c,r,fc,fr)} \neq \emptyset$, there is a path $pa = (cs_1^0, \ldots, cs_{n+1}^0), (c_1^0, \ldots, c_{n+1}^0))$ $\xrightarrow{act_1} \ldots \xrightarrow{act_w} ((cs_1^w, \ldots, cs_{n+1}^w), (c_1^w, \ldots, c_{n+1}^w))$ of $M_1^{k\text{-}(f,exth)} \otimes \ldots \otimes M_{n+1}^{k\text{-}(f,exth)}$, such that for each process id i, $cs_i^0 = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon)$, $c_i^0 = \epsilon$, $cs_i^w = (p_w(i), d_w, d_w, q_{ends}, \epsilon)$ and $cs_i^w = \epsilon$. Let $cs_i^j = (q_i^j, d_{ci}^j, d_{gi}^j, q_{si}^j, mak_i^j, cnt_i^j)$ for each process id i.

We prove this lemma by constructing a path pa' of $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, such that the call, return, flushCall and flushReturn in *pa* is same as that of pa'.

A relation $\sim$ is defined as follows: given configuration $((cs_1^v, \ldots, cs_{n+1}^v), (c_1^v, \ldots, c_{n+1}^v))$ for the *v+1-th* configuration of the path, and a configuration (p, d, u, mak) of $[\![Clt_f(Mod(\ \mathcal{L})), n{+}1]\!]_b$, $((cs_1^v, \ldots, cs_{n+1}^v), (c_1^v, \ldots, c_{n+1}^v)) \sim (p, d, u, mak)$, if,

- For each process id i, $q_i^v = p(i)$, $d_{ci}^v = d$, $mak_i^v = mak$.
- For each process id i_1, i_2, $q_{si_1}^v = q_{si_2}^v$, $mak_{i_1}^v = mak_{i_2}^v$, $cnt_{i_1}^v = cnt_{i_2}^v$.
- Let $c_i^{v'}$ be generated from c_i^v by discarding items of all but process i. Then for each *ind*, $u(ind) = (x, a)$, if and only if $c_i^{v'}(ind) = (i, x, val)$ or $((i, x, val), \sharp)$ for some *val* where $val(x) = a$.

It remains to prove that if $((cs_1^v, \ldots, cs_{n+1}^v), (c_1^v, \ldots, c_{n+1}^v)) \sim (p, d, u, mak)$ and $((cs_1^v, \ldots, cs_{n+1}^v), (c_1^v, \ldots, c_{n+1}^v)) \xrightarrow{act_{v+1}} ((cs_1^{v+1}, \ldots, cs_{n+1}^{v+1}), (c_1^{v+1}, \ldots, c_{n+1}^{v+1}))$, then one of the following two cases holds:

- Case 1: there exists configuration (p', d', u', mak'), such that $((cs_1^{v+1}, \dots, cs_{v+1}^{n+1}), (c_1^{v+1}, \dots, c_{n+1}^{v+1})) \sim (p', d', u', mak')$, $(p, d, u, mak) \xrightarrow{s_{v+1}}{}_b^{*} (p', d', u', mak')$, the flush, call, return, flushCall and flushReturn action in act_{v+1} are same to that in s_{v+1},
- Case 2: $((cs_1^{v+1}, \dots, cs_{n+1}^{v+1}), (c_1^{v+1}, \dots, c_{n+1}^{v+1})) \sim (p, d, u, mak)$.

We prove this by considering all kinds of transition label act_{v+1},

- If act_{v+1} is a internal action derived from a τ or read action of some process, then $s_{v+1} = \epsilon$ and case 1 holds trivially.
- If act_{v+1} is a call, return, flushCall or flushReturn action, then it is easy to see that $s_{v+1} = act_{v+1}$, (p', d', u', mak') is generated from (p, d, u, mak) by a act_{v+1} transition, and case 1 holds.
- If act_{v+1} is a internal action derived from a $write(i, x, val)$ transition of $M_i^{k\text{-}(f,exth)}$, then $s_{v+1} = write(i, x, val(x))$, case 1 holds, (p', d', u', mak') is generated from (p, d, u, mak) by $write(i, x, val(x))$ transition.
- If act_{v+1} is a internal action derived from a guessing write transition of $M_i^{k\text{-}(f,exth)}$, then it is obvious that case 2 holds.
- When act_{v+1} is a flush action derived from a $flush(i, x, val)$ transition of $M_i^{k\text{-}(f,exth)}$, then $s_{v+1} = flush(i, x, val(x))$, case 1 holds, (p', d', u', mak') is generated from (p, d, u, mak) by $flush(i, x, val(x))$ transition.
- When act_{v+1} is a flush action derived from a $cas(i, x, val, val')$ transition of $M_i^{k\text{-}(f,exth)}$, then $s_{v+1} = cas(i, x, val(x), val'(x))$, case 1 holds and (p', d', u', mak') is generated from (p, d, u, mak) by a $cas(i, x, val(x), val'(x))$ transition.

We construct pa' step by step, and it is obvious that the extended history of pa' is eh, which completes the proof. □

A.4 Proof of Lemma 10

A configuration $((q, d_c, d_g, q_s, mak, cnt), c)$ of $M_i^{k\text{-}w}$ is called standard, if one of the following two conditions holds: (1) $c = \epsilon \wedge d_c = d_g$, (2) $c \neq \epsilon$, $c(1)$ is a strong symbol and $c(1) = (\_, \_, d_g)$ or $((\_, \_, d_g), \sharp)$. It is obvious if a path of (S, K_1)-(lossy) channel machine starts from a standard configuration, then each configuration on this path is standard.

The following lemma shows that there is a weak simulation between configurations of (S, K_1)-channel machine $M_i^{k\text{-}w}$ and configurations of (S, K_1)-lossy channel machine $M_i^{k\text{-}w}$.

Lemma 15. *Given standard configuration* $((p_1, d_{c1}, d_{g1}, q_s^1, mak_1, cnt_1), c_1)$*, if* $c_1 \preceq_S^{K_1} c_1'$ *and* $((p_1, d_{c1}, d_{g1}, q_s^1, mak_1, cnt_1), c_1) \xrightarrow{act}_{(M_i^{k\text{-}w}, S, K_1)} ((p_2, d_{c2}, d_{g2}, q_s^2, mak_2, cnt_2), c_2)$*, then there exists* c_2' *and* s*, such that* $c_2 \preceq_S^{K_1} c_2'$*,* $((p_1, d_{c1}, d_{g1}, q_s^1, mak_1, cnt_1), c_1') \xrightarrow{s}{}_{M_i^{k\text{-}w}}^{*} ((p_2, d_{c2}, d_{g2}, q_s^2, mak_2, cnt_2), c_2')$*, and the write actions in act equals that in* s*.*

Proof. This is proved by considering all kinds of transitions.

- If act is a internal action derived from a τ, read, call or return action, then $c_2' = c_1'$ and this holds trivially.

- If *act* is a write action derived from a *cas* action, then $c'_2 = \epsilon$ and this holds trivially.
- If *act* is a write action derived from a *write*(ind, x, val) action, then c'_2 is generated from c'_1 by a write action that puts an item of memory location x and valuation *val*, and this holds trivially.
- If *act* is a internal action derived from a flush action, assume $c_1 = \alpha_1 \cdot \ldots \cdot \alpha_l$, $c'_1 = \beta_1 \cdot \ldots \cdot \beta_w$, since $c_1 \preceq_S^{K_1} c'_1$, there exists $i_1, \ldots, i_l$, such that for each *ind*, $\alpha_{ind} = \beta_{i_{ind}}$.
  Assume during the transition $((p_1, d_{c1}, dg1, q_s^1, mak_1, cnt_1), c_1) \xrightarrow{\alpha}_{(M_i^{k\text{-}w}, S, K_1)} ((p_2, d_{c2}, dg2, q_s^2, mak_2, cnt_2), c_2)$, the item which is flushed into memory is the *j-th* element in c_1. Then $s = \epsilon$, and $((p_2, d_{c2}, dg2, q_s^2, mak_2, cnt_2), c'_2)$ is generated from $((p_1, d_{c1}, dg1, q_s^1, mak_1, cnt_1), c'_1)$ by first flushing items $\beta_w, \ldots, \beta_{i_j+1}$, and then flush item β_{i_j}.

□

With Lemma 15, we can now prove Lemma 10.

Lemma 10. *Given a k-extended history eh. There exists a marked witness t of eh from $(p_{in}, d_{init}, \epsilon^{n+1}, \epsilon)$ to $(p_w, d_w, \epsilon^{n+1}, \epsilon)$ in $[\![Clt_f(Mod(\mathcal{L})), n{+}1]\!]_b$, if and only if $\cap_{i=1}^{n+1} LT_{(q_i, q'_i)}^{(S, K_1)} M_i^{k\text{-}w} \neq \emptyset$, where for each $1 \leq i \leq n{+}1$, $q_i = (p_{in}(i), d_{init}, d_{init}, q_{is}, \epsilon, 0)$, $q'_i = (p_w(i), d_w, d_w, q_{ends}, \epsilon, |t \uparrow_{(\Sigma_{cal} \cup \Sigma_{ret} \cup \Sigma_{fcal} \cup \Sigma_{fret})}|)$.*

Proof. According to Lemma 9, we reduce the existence of t into $\cap_{i=1}^{n+1} T_{(q_i, q'_i)}^{(S, K_1)} M_i^{k\text{-}w} \neq \emptyset$. Then, we need to prove that, $\cap_{i=1}^{n+1} T_{(q_i, q'_i)}^{(S, K_1)} M_i^{k\text{-}w} \neq \emptyset$, if and only if $\cap_{i=1}^{n+1} LT_{(q_i, q'_i)}^{(S, K_1)} M_i^{k\text{-}w} \neq \emptyset$. The *only if* direction is obvious, since a sequence $l \in \cap_{i=1}^{n+1} T_{(q_i, q'_i)}^{(S, K_1)} M_i^{k\text{-}w}$ is also a sequence of $\cap_{i=1}^{n+1} LT_{(q_i, q'_i)}^{(S, K_1)} M_i^{k\text{-}w}$. To prove the *if* direction, by Lemma 15, we can construct a path of perfect channel from a path of lossy channel step by step, where both path has same write actions. This completes the proof of this lemma. □