Analyzing Mutable Checkpointing via Invariants

Deepanker Aggarwal, Astrid Kiehn

Analyzing Mutable Checkpointing via Invariants

Deepanker Aggarwal and Astrid Kiehn

Indraprastha Institute of Information Technology,
New Delhi, India
{deepanker10027,astrid}@iiitd.ac.in
http://www.iiitd.ac.in

Abstract. The well-known coordinated snapshot algorithm of mutable checkpointing [7–9] is studied. We equip it with a concise formal model and analyze its operational behavior via an invariant characterizing the snapshot computation. By this we obtain a clear understanding of the intermediate behavior and a correctness proof of the final snapshot based on a strong notion of consistency (reachability within the partial order representing the underlying computation). The formal model further enables a comparison with the blocking queue algorithm [13] introduced for the same scenario and with the same objective.
From a broader perspective, we advocate the use of formal semantics to formulate and prove correctness of distributed algorithms.

Key words: snapshot, checkpointing, consistency, distributed computing

1 Introduction

The on-the-fly calculation of a snapshot, a consistent global state, is a known means to enhance fault tolerance and system diagnosis of distributed systems. Coordinated snapshot algorithms exchange coordination messages to orchestrate the checkpointing. One of these is mutable checkpointing ([7–9]) which aims at a reduced coordination overhead - both in number of checkpoints to be taken and coordination messages to be sent. It is known from [7] that there is no algorithm which minimizes the number of checkpoints without blocking processes. To avoid the blocking, in mutable checkpointing, local checkpoints may be taken which may be discarded later. The presence of such checkpoints and an additional feature to further reduce the number of coordination messages hinder an easy analysis of the algorithm. With this paper we equip mutable checkpointing with a precise formal model and make it amenable to a formal analysis. We establish an invariant to obtain deeper insight into the intermediate behavior of the algorithm and prove consistency of the final snapshot. The model can further be used as a common ground for qualitative comparisons of snapshot algorithms. We give such a comparison with the conceptually different blocking queue algorithm [13] which, as mutable checkpointing, had been set up to reduce the coordination overhead.

Unlike other coordinated snapshot algorithms (eg. Chandy/ Lamport's seminal algorithm [10]), mutable checkpointing algorithms do not monotonously build up the snapshot with the underlying computation. In mutable checkpointing, checkpoints of local states may be taken from which the underlying computation had already progressed. The computation of the snapshot, thus, involves forward and backward reasoning and the correctness of the algorithm, the consistency of the final snapshot, is not obvious. The proofs provided in the literature [7–9] are based on contradiction, use absence of orphans (messages recorded as received but not as being sent) as consistency notion, and lack a formal model. The formal proof provided in this paper is based on an invariant which characterizes the snapshot partially computed. We use a stronger but well-accepted notion of consistency [17]: reachability within the partial order representing the underlying distributed computation. It implies the absence of orphans.

In case of a snapshot algorithm the invariant should explain how the snapshot gradually builds up on course of the underlying computation. For a global state S of the underlying computation, the invariant should provide the snapshot calculated so far. We call the latter the *potential snapshot* $psn(S)$ of S. For a snapshot algorithm which simply freezes local processes at certain points of their computation, the potential snapshot consists of these checkpoints and the current states of the none-frozen processes. In mutable checkpointing, local processes may be frozen to states – the so-called mutable checkpoints - from which they had already progressed. Reachability of $psn(S)$ can therefore not simply be obtained from a simultaneous progression of $psn(S)$ and S. We solve this problem by extracting a set of global states from S of which each corresponds to a different prediction of which of the mutable checkpoints will be frozen. These states collectively define the $psn(S)$. We then show how each of the states in $psn(S)$ progresses together with S, where the progress may be partial, only, due to the frozen processes. Using this result we provide a direct proof for the consistency of the final snapshot. The potential snapshot, however, – or more precisely the predicted states of it -, in general, are shown not to be consistent. This implies that if the run of a mutable checkpointing algorithm needs to be interrupted then the entire checkpointing needs to be started afresh.

The proof is based on the specification of the operational behavior of what we consider the essence of mutable checkpointing. We extracted it from the pseudocode given in [9] by removing all details not related to the basic concept of taking a mutable checkpoint upon receiving a flagged message for the first time (and before a checkpoint) – where the flag indicates that the sender had taken a checkpoint or a mutable checkpoint. In this way we obtained a concise description of the core of mutable checkpointing which we see as another contribution of this paper.

With this formal model and analysis we relate mutable checkpointing to the blocking queue algorithm introduced in [13]. In fact, this paper can be seen as a companion paper as it deploys the proof technique developed there (however, setting up the invariant for mutual checkpointing was a much more demanding task). Having fixed the underlying computation, the two snapshot algorithms can directly be compared due to the same underlying formal model. We show that the respective final snapshots, in general, are incomparable and discuss the differences between the algorithms.

The paper is structured as follows. Basic terminology is introduced in Section 2, followed by a short description of mutable checkpointing in Section 3. Section 4 specifies the operational behavior of the algorithm defined in terms of predicates and rules. The rules we deduced from the pseudocode of [9] where we abstracted away as many details as possible to get the essence of mutable checkpointing. In Section 5 we introduce the potential snapshot, show its progression with the underlying computation and prove the consistency of the final snapshot. Section 6 gives the comparison with the blocking queue algorithm of [13]. The conclusion is given in Section 7.

2 Preliminaries

We assume a finite number of processes P_1, ..., P_n which communicate solely by message passing via FIFO channels C_{ij}. Channel C_{ij} leads from P_i to P_j and for each pair of processes there is such a unidirectional channel. Channels are assumed not to lose or reorder messages. There is no assumption on the state space of processes.

A state $S = (p_1, \ldots, p_n, \mathit{Chan})$ of a distributed computation is given by the histories (events performed so far) of the local processes and the current contents of the channels where $\mathit{Chan} : \{C_{ij} \mid i, j \leq n, i \neq j\} \to \mathit{MSG}^*$ and $p_i \in \mathit{Events}^*$. To ease readability, for a global state S we attach S as a superscript to the histories and channels and abbreviate $\mathit{Chan}(C_{ij})^S$ by C_{ij}^S. In the initial state S_0, $p_i^{S_0} = \varepsilon$, and $C_{ij}^{S_0} = \varepsilon$ for all i, j. A distributed computation is a sequence $\pi = S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \cdots \xrightarrow{e_k} S_k$ where each S_i is obtained from updating S_{i-1} according to the semantics of event e_i. We also write $\pi = S_0 \to^* S_k$ to mention the initial and final states of π, explicitly.

The notion of consistency of a state with a computation π is best understood in terms of π's space-time diagram (its partial order representation, see [14], [17] or [4]). A state is consistent with π if it is a cut of π closed under the *happens-before* ordering in the space-time diagram. Equivalently, a consistent state S of π can be characterized by that all local histories of S are prefixes of the corresponding histories of π's final state S_k, and if a message occurs as received in a history of S then it also needs to occur as having been sent in a history (that is there are no orphans).

Snapshot algorithms are superimposed on a distributed computation π on course of which a state consistent with π is to be calculated. We will only describe the behavior of the snapshot algorithm which, if it is non blocking, should allow for send and receive events induced by the underlying computation at any point of time. All other events of the snapshot algorithm are coordination events.

3 Mutable Checkpoint Algorithms

Mutable checkpoint algorithms are coordinated snapshot algorithms which reduce the coordination overhead (compared to [10]) by combining message flags (indicating whether messages have been sent before of after a checkpoint, cf. [15]) with the new concept of mutable checkpoints. Mutable checkpoints are taken on a tentative basis (on volatile storage) and are only finalized (on non-volatile storage) when the need for a local checkpoint has been confirmed.

The Algorithm in Short

The initiating process requests the processes it depends on (from which it had received a message) to take a checkpoint. Any process receiving such a request takes a checkpoint and propagates the request further to the – up to its knowledge – so far uninformed processes it itself depends on. After a process has taken a checkpoint all the messages sent out by this process carry a flag (bb=1). A process which hasn't received a checkpoint request but a message with flag (bb=1), takes a mutable checkpoint indicating that it must convert the current local state to a checkpoint, if in future it receives a checkpoint request. This is done before the received message is processed and only if it hadn't taken a mutable checkpoint earlier. Under certain progress assumptions, all processes which, in principle, need to take a checkpoint will finally have done so and this completes phase I of the algorithm. Phase II would deal with the confirmation that the checkpointing is complete and the dissemination of this information to the local processes. However, in this paper we only investigate phase I.

With minor modifications the algorithm has widely been published see [7–9], our reference algorithm is [9]. We specify the operational behavior of the algorithm in terms of predicates and transition rules which an implementation would need to satisfy. To be able to focus on the essence of mutable checkpointing, in the translation we omitted everything related to earlier checkpointing and a feature to reduce the number of coordination messages further (the $sent_i$ condition). We also assume that the initiating process is always P_1 and as in [9] do not consider concurrent checkpointing. Finally, all details relating to termination (the completion of taking checkpoints) of phase I are omitted.

The Algorithm in Detail

Rule numbers in brackets refer to corresponding rules given in the next section.

1. As part of their computations, processes send messages to each other which come attached with a flag (Rules 1.1 and 1.2). If the flag is set, this indicates that the sending process has already taken its checkpoint (instantly or belated via a mutable checkpoint).

2. Every process maintains a dependency vector which provides all the processes it depends on. A process P_i depends on process P_j if P_i has received a message from P_j. This is a dynamic notion of dependency as at the time of initiation of the checkpointing all dependencies may not be known. The checkpointing will involve all processes which are dependent in a transitive way. Say at the time of initiation, the checkpointing process P_1 depends on P_2 and P_3, and P_3 depends on P_4. Then, P_4 needs to be included in the checkpointing.
3. The initiator takes its checkpoint and sends the checkpoint request to the processes it depends on using its dependency vector. It attaches the dependency vector to its request (Rule 3).
4. If a process P_j receives a checkpoint request from a process P_i then either of the following will happen:
   - If it has already taken a checkpoint (cp_taken_j is true), then the request is ignored (Rule 4.1).
   - If it has not taken a checkpoint but has taken a mutable checkpoint (mcp_taken_j is true), then by receiving the request it converts the mutable checkpoint into a checkpoint. This conversion is not explicitly modeled but from now on cp_taken_j will be true. Further, P_j propagates the checkpoint request to processes as follows. For each process P_k on which P_i does not depend on, but on which P_j depended when it took the mutable checkpoint, P_j sends a request to P_k (P_i has already sent a request to the processes on which it depends, Rule 4.2).
   - If it has neither taken a checkpoint or a mutable checkpoint, then it takes a checkpoint and propagates the request as in the previous case (Rule 4.3).
5. If a process P_j removes a message (with attached flag) from a channel then either of the following will happen:
   - If the received message has flag 0, then P_j processes the message (Rule 2.1, Rule 2.2). If this happens before a mutable checkpoint or checkpoint is taken, then P_j depends on P_i and the dependency vector might need to be updated (Rule 2.1).
   - If the flag is 1 and P_j has already taken a checkpoint, then P_j processes the message (Rule 2.3).
   - If the flag is 1 and P_j has neither taken a checkpoint nor mutable checkpoint, then it takes a mutable checkpoint and immediately after that processes the message (Rule 2.4). Taking the mutable checkpoint and processing the message is one atomic action.
   - If the flag is 1 and P_j has taken a mutable checkpoint but not a checkpoint, then P_j processes the message (Rule 2.5).

4 The Operational Behavior of Mutable Checkpointing

We specify the algorithm's behavior by a set of predicates and rules describing how the global state of the system changes with a transition. The rules of the following format:

Rule No.	Preconditions	Event	Postconditions

If a global state S satisfies the precondition of a rule, then the event may occur and S is updated to T as specified in the field of postconditions. The occurrence of event e is written as $S \xrightarrow{e} T$. So a distributed computation $\pi = S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \cdots \xrightarrow{e_k} S_k$ is a sequence of such events where each of the transitions is justified by one of the rules. In some of the rules (Rules 1.2 and 2.2) the preconditions are split into two rows. These should be read as a disjunction, that is, each of these rules presents two rules with the same event name and postconditions. The possible events and messages of the mutable checkpointing algorithm are given by the following table.

mcp_taken_i	process P_i takes a mutable checkpoint
cp_taken_i	P_i takes the checkpoint
$send_{ij}(.)$	P_i adds a message to channel C_{ij}
$rec_{ij}(.)$	P_j receives a message or checkpoint request from channel C_{ij}
$\langle cpr_i, dep\rangle$	the message that P_i has taken a checkpoint with attached dependency vector
$\langle msg, bb\rangle$	a message and attached flag

In the algorithm every local process maintains a dependency vector dep in which it keeps the dependencies to other processes: $dep_j(i) = 1$ if P_j has received a message from P_i before a mutable checkpoint has been taken. This dependency vector can be retrieved from the history of a process at any state. However, for clarity we explicitly mention it in the semantics.

The first element of a channel is at the rightmost position and provided by *first* and the remainder by *rem*. We use the simple dot to separate letters in a word. For the concatenation of words we use $\circ$. If an event occurs in the history of a process at state S then we state this as a predicate $event_i^S$. For example, $\neg mcp\_taken_i^S$ stands for that mcp_taken_i does not occur in the history p_i^S. It represents that P_i has not taken a mutable checkpoint so far. The $cp\_taken_i^S$ predicate, however, is more general as it also needs to cover the conversion of a mutable checkpoint to a (proper) checkpoint. Hence, $cp\_taken_i^S$ if and only if either cp_taken_i occurs in the history of P_i or $mcp\_taken_i^S$ and $rec_{ji}(\langle cpr_j, dep\rangle)^S$ for some j.

Rules are grouped according to their functionality.

5 Main Results

We here define the potential snapshot and show how it progresses with the underlying computation. With this invariant result we will show the reachability of the snapshot finally calculated.

As already discussed, in mutable checkpointing the potential snapshot $psn(S)$ extracted from an intermediate state S of the underlying computation cannot simply be a global state containing the current local checkpoints. At S there is

No	Preconditions	Event	Postconditions
1.1	$\neg cp\_taken_i^S$ $\neg mcp\_taken_i^S$	$S \xrightarrow{send_{ij}(\langle msg,0\rangle)} T$	$p_i^T = p_i^S.send_{ij}(\langle msg,0\rangle)$ $C_{ij}^T = \langle msg,0\rangle.C_{ij}^S$
1.2	$cp\_taken_i^S$	$S \xrightarrow{send_{ij}(\langle msg,1\rangle)} T$	$p_i^T = p_i^S.send_{ij}(\langle msg,1\rangle)$ $C_{ij}^T = \langle msg,1\rangle.C_{ij}^S$
	$mcp\_taken_i^S$		
2.1	$first(C_{ij}^S) = \langle msg,0\rangle$ $\neg mcp\_taken_j^S$ $\neg cp\_taken_j^S$	$S \xrightarrow{rec_{ij}(\langle msg,0\rangle)} T$	$p_j^T = p_j^S.rec_{ij}(\langle msg,0\rangle)$ $C_{ij}^T = rem(C_{ij}^S)$ $dep_j^T(i) = 1$
2.2	$first(C_{ij}^S) = \langle msg,0\rangle$ $mcp\_taken_j^S$	$S \xrightarrow{rec_{ij}(\langle msg,0\rangle)} T$	$p_j^T = p_j^S.rec_{ij}(\langle msg,0\rangle)$ $C_{ij}^T = rem(C_{ij}^S)$
	$first(C_{ij}^S) = \langle msg,0\rangle$ $cp\_taken_j^S$		
2.3	$first(C_{ij}^S) = \langle msg,1\rangle$ $cp\_taken_j^S$	$S \xrightarrow{rec_{ij}(\langle msg,1\rangle)} T$	$p_j^T = p_j^S.rec_{ij}(\langle msg,1\rangle)$ $C_{ij}^T = rem(C_{ij}^S)$
2.4	$first(C_{ij}^S) = \langle msg,1\rangle$ $\neg cp\_taken_j^S$ $\neg mcp\_taken_j^S$	$S \xrightarrow{rec_{ij}(\langle msg,1\rangle)} T$	$p_j^T = p_j^S.mcp\_taken_j.rec_{ij}(\langle msg,1\rangle)$ $C_{ij}^T = rem(C_{ij}^S)$
2.5	$first(C_{ij}^S) = \langle msg,1\rangle$ $\neg cp\_taken_j^S$ $mcp\_taken_j^S$	$S \xrightarrow{rec_{ij}(\langle msg,1\rangle)} T$	$p_j^T = p_j^S.rec_{ij}(\langle msg,1\rangle)$ $C_{ij}^T = rem(C_{ij}^S)$
3	$\neg cp\_taken_1^S$	$S \xrightarrow{cp\_taken_1} T$	$p_1^T = p_1^S.cp\_taken_1$ $C_{1k}^T = \langle cpr_1, dep_1^S\rangle.C_{1k}^S$ for all $k > 1$ with $dep_1^S(k) = 1$
4.1	$first(C_{ij}^S) = \langle cpr_i, dep\rangle$ $cp\_taken_j^S$	$S \xrightarrow{rec_{ij}(\langle cpr_i, dep\rangle)} T$	$p_j^T = p_j^S.rec_{ij}(\langle cpr_i, dep\rangle)$ $C_{ij}^T = rem(C_{ij}^S)$
4.2	$first(C_{ij}^S) = \langle cpr_i, dep\rangle$ $\neg cp\_taken_j^S$ $\neg mcp\_taken_j^S$	$S \xrightarrow{cp\_taken_j} T$	$p_j^T = p_j^S.rec_{ij}(\langle cpr_i, dep\rangle).cp\_taken_j$ $C_{ij}^T = rem(C_{ij}^S)$ $C_{jk}^T = \langle cpr_j, dep \vee dep_j^S\rangle.C_{jk}^S$ for all k with $dep(k) = 0$, $dep_j^S(k) = 1$
4.3	$first(C_{ij}^S) = \langle cpr_i, dep\rangle$ $\neg cp\_taken_j^S$ $mcp\_taken_j^S$	$S \xrightarrow{rec_{ij}(\langle cpr_i, dep\rangle)} T$	$p_j^T = p_j^S.rec_{ij}(\langle cpr_i, dep\rangle)$ $C_{ij}^T = rem(C_{ij}^S)$ $C_{jk}^T = \langle cpr_j, dep \vee dep_j^S\rangle.C_{jk}^S$ for all k with $dep(k) = 0$, $dep_j^S(k) = 1$

Table 1. Rules 1.1, 1.2: A message can be sent at any time and the attached flag shows whether this happened before (bb=0) or after (bb=1) a checkpoint or mutable checkpoint was taken. Rules 2.1–2.5: A mutable checkpoint is taken if the flag of the received message is 1 and the receiving process has neither taken a checkpoint nor a mutable checkpoint so far. Rule 3: We assume that the checkpointing will always be initiated by P_1. It sends the checkpoint request to all the processes it depends on and takes the checkpoint as part of one atomic action. Rules 4.1–4.3: Receiving, setting and propagating a checkpoint request is modeled as one atomic event. This event comprises of removing the request from the channel, taking the checkpoint and propagating the request and causal dependencies to the concerned processes. These are the processes on which the receiving process depends but which are not listed in the dependency array received with the incoming request. In case a mutable checkpoint had been taken, it is converted to a permanent one (this, however, is not explicitly modeled).

no clarity whether a mutable checkpoint should be considered as a checkpoint or simply be discarded since the future computation steps cannot be foreseen. In $psn(S)$ all options have to be simultaneously considered. Accordingly, $psn(S)$ is a set of global states of which each corresponds to a different prediction with respect to the final conversion of mutable checkpoints to (proper) checkpoints.

The formal definitions are given next. Note that all projection functions used in this paper are summarized in Figure 1. The freeze function allows one to cut down the history of individual processes to the point where they have taken a mutable checkpoint or a checkpoint. These points we call freeze points. In order to freeze a process P_i all events after the freeze point need to be deleted from the history. In general, the freezing of P_i may effect process P_j as the latter may have received a message from P_i sent after its freeze point. Processes that may be frozen are those in $MCP(S)$ while those in $CP(S)$ must be frozen if a freezing is to be conducted.

$$\begin{aligned} MCP(S) &= \{P_i \mid mcp\_taken_i^S \text{ and } \neg cp\_taken_i^S\} \\ CP(S) &= \{P_i \mid cp\_taken_i^S\} \end{aligned}$$

Due to the various freezing options we obtain the set of potential snapshots:

$$PSN(S) = \{f(A,S) \mid CP(S) \subseteq A \subseteq MCP(S) \cup CP(S)\}$$

Each $f(A,S)$ provides a potential snapshot at state S of which the processes in A are frozen.

$$f(A,S) = < p_1^{f(A,S)}, p_2^{f(A,S)}, \ldots, p_n^{f(A,S)}, Chan^{f(A,S)} >$$

$$p_i^{f(A,S)} = \begin{cases} p_i^S \downarrow_{mcp,cp} & P_i \in A \\ p_i^S|_{rem\, bb=1}^A & P_i \notin A \end{cases}$$

$$Chan^{f(A,S)} = \{C_{ij}^{f(A,S)} \mid 1 \leq i,j \leq n \text{ and } i \neq j\}$$

$$C_{ij}^{f(A,S)} = \begin{cases} \widehat{C_{ij}^S|_{bb=0}} \text{ o } rev(p_j \uparrow^i_{mcp,cp,bb=0}) & P_i, P_j \in A \\ \widehat{C_{ij}^S|_{bb=0}} & P_i \in A,\, P_j \notin A \\ \widehat{C_{ij}^S} \text{ o } rev(p_j \uparrow^i_{mcp,cp}) & P_i \notin A,\, P_j \in A \\ \widehat{C_{ij}^S} & P_i \notin A,\, P_j \notin A \end{cases}$$

where $\widehat{C_{ij}}$ removes all the coordination messages from the channel and rev reverses the string. Note that we do not explicitly remove the flag (which are used for coordination reasons only) from a message, but this is implied whenever coordination messages are removed.

To show the simultaneous progression of $psn(S)$ with $S \xrightarrow{e} S'$ each of the states in $psn(S)$ needs to be considered with respect to its corresponding move. If a mutable checkpoint has to be frozen then those states in $psn(S)$ which had not predicted this are discarded. This means that the size of $psn(S)$ can shrink. It will, however, grow with every new mutable checkpoint taken as the

$\downarrow_{mcp,cp}$ It is applied to the local history of a process and yields the string of send and receive events before *mcp_taken* or *cp_taken* in their respective order. Only one of the latter events can occur in a history.

$|^{A}_{rem\, bb=1}$ It is applied to the local history of a process not in A and removes all coordination messages and all messages with flag 1 received from a process in A.

$\uparrow^{i}_{mcp,cp}$ It is applied to the local history of a process P_j and yields the string of messages in $rec_{ij}(\langle msg, bb\rangle)$ events after *mcp_taken* or *cp_taken* in their respective order. Only one of the latter events can occur in a history.

$\uparrow^{i}_{mcp,cp,bb=0}$ It is applied to the local history of a process P_j and yields the string of messages in $rec_{ij}(\langle msg, 0\rangle)$ events after *mcp_taken* or *cp_taken* in their respective order. Only one of the latter events can occur in a history.

$|^{j}_{sent}$ It is applied to the local history of a process P_i and yields the string of messages occurring in $send_{ij}(\langle msg, bb\rangle)$ events in their respective order.

$|_{bb=0}$ This projection is applied to strings of messages, only. It removes from the string all coordination messages and messages with attached flag 1.

Fig. 1. Projection functions used in this paper.

concerned process will now have to be considered as progressing and as frozen, simultaneously. The next lemma describes this progression in detail.

Lemma 1. *Let $S_0 \longrightarrow^* S$ and $S \xrightarrow{e} S'$ where e is an event of P_i, $i \in \{1, \ldots, n\}$. Then for all freeze sets A of S the following holds:*

1. *$P_i \in A$ implies $f(A, S) = f(A, S')$*
2. *$P_i \notin A$ implies one of the following:*
   (a) *$e \notin \{cp\_taken_i, rec_{ji}(\langle cpr_j, dep\rangle), rec_{ji}(\langle msg, 1\rangle) \mid j \in \{1, \ldots, n\}\}$, and $f(A, S) \xrightarrow{e} f(A, S')$*
   (b) *$e = rec_{ji}(\langle cpr_j, dep\rangle)$, $mcp\_taken^S_i$, and $f(A \cup \{P_i\}, S) = f(A \cup \{P_i\}, S')$*
   (c) *$e = cp\_taken_i$, $\neg mcp\_taken^S_i$, and $f(A, S) = f(A \cup \{P_i\}, S')$*
   (d) *$e = rec_{ji}(\langle msg, 1\rangle)$, $P_j \notin A$, and*
      i. *$f(A, S) \xrightarrow{e} f(A, S')$*
      ii. *$f(A, S) = f(A \cup \{P_i\}, S')$ if $\neg mcp\_taken^S_i$*
   (e) *$e_i = rec_{ji}(\langle msg, 1\rangle)$, $P_j \in A$, and*
      i. *$f(A, S) = f(A, S')$*
      ii. *$f(A, S) = f(A \cup \{P_i\}, S')$ if $\neg mcp\_taken^S_i$*
3. *$p_k^{f(A', S')} = p_k^{S'}$ if $P_k \notin A'$, $\neg\, mcp\_taken^{S'}_k$, $\neg\, cp\_taken^{S'}_k$ and A' is a freeze set of S'*

where $A \cup \{P_i\}$ is a freeze set of the respective state whenever given as an argument to the freeze function.

Proof. By induction on the number of transitions leading from S_0 to S.
If there is no transition then $S_0 = S$ and $A = \emptyset$. Of item 2. only 2.(a) and 2.(c) apply which can be easily verified, and item 3. is trivial.
Now suppose the induction hypothesis applies to $S_0 \rightarrow^* S$ and there is one more

transition $S \xrightarrow{e} S'$. We have to explore all freeze sets of the relevant state and all the possible transitions.

We sketch two cases, for all others we refer to [2].
Proof of item 2.(c) for Rule 4.2.
Rule 4.2 deals with the case $e = cp\_taken_i$, $first(C_{ji}^S) = \langle cpr_j, dep\rangle$, $\neg cp\_taken_i^S$ and $\neg mcp\_taken_i^S$. Let A be a freeze set of S.

$$\begin{aligned} p_i^{f(A\cup\{P_i\},S')} &= p_i^{S'} \downarrow_{mcp,cp} \\ &= p_i^S.rec_{ji}(\langle cpr_j, dep\rangle.cp\_taken_i) \downarrow_{mcp,cp} \\ &= p_i^S && \text{def. } \downarrow_{mcp,cp} \\ &= p_i^{f(A,S)} && \text{ind. hyp. 3.} \end{aligned}$$

The proofs for the other p_k, $k \neq i$, and the channels are similar.
Proof of item 3. for Rule 4.2.
Let A' be a freeze set of S'. Then $A' \setminus \{P_i\}$ is a freeze set of S, and by induction hypthesis 2.(c), $f(A' \setminus \{P_i\}, S) = f(A', S')$. Let $P_k \notin A'$, $\neg\, mcp\_taken_k^S$ and $\neg\, cp\_taken_k^S$.

$$\begin{aligned} p_k^{f(A',S')} &= p_k^{f(A'\setminus\{P_i\},S)} && \text{ind. hyp. 2.(c), already established} \\ &= p_k^S && \text{ind. hyp. 3.} \\ &= p_k^{S'} \end{aligned}$$

Note again, that in the lemma notationally we did not distinguish between a message with or without flag. However, in all events performed by a frozen process the flags have been removed. Similarly, in the next lemma $\sqsubseteq$ denotes the prefix relation up to messages with or without flags. We further do not distinguish between S_0 and $f(\emptyset, S)$. Lemma 2 summarizes the invariant property of reachable states relevant for the consistency proof of the final snapshot. It is an immediate corrollary of Lemma 1 and the basic definitions.

Lemma 2. *Let $\pi : S_0 \to^* S$.*
If A is a freeze set of S then $S_0 \to^* f(A,S)$ and
(1) $p_k^{f(A,S)} \sqsubseteq p_k^S$ *for $P_k \in A$,*
(2) $p_k^{f(A,S)} = p_k^S$ *for $P_k \notin A$, $\neg\, mcp\_taken_k^S$ and $\neg\, cp\_taken_k^S$,*
(3) $p_k^{f(A,S)} = p_k^S|_{rem\, bb=1}^A$ *for $P_k \notin A$, $mcp\_taken_k^S$ or $cp\_taken_k^S$.*

Lemma 2 shows that each $f(A,S)$ is reachable. This, however, does not mean that $f(A,S)$ is consistent with π since events of frozen processes are removed from the history and may create "holes". One may argue that the states to be considered here should be those with all none-frozen processes reset to their initial states. This would avoid the "hole" problem. For an arbitrary set of processes I, $I \subseteq \{P_1, \ldots, P_n\}$, the reset of I at S is defined by

$$p_i^{r(I,S)} = \begin{cases} p_i^S & \text{if } P_i \notin I, \\ \varepsilon & \text{otherwise} \end{cases} \quad \text{and}$$

$$C_{ij}^{r(I,S)} = \begin{cases} C_{ij}^S & P_i, P_j \notin I, \\ p_i^S|_{sent}^j & P_j \in I, \\ \varepsilon & P_i \in I. \end{cases}$$

In general, $r(I, S)$ is not a consistent state. This is always the case if a process in I had sent a message to a process not in I and this message had been received before the resetting. It would become an orphan.

This problem does not occur if S is the final state of a computation with completed checkpointing. The checkpointing is complete if there is no coordination message in any of the channels (recall that taking a checkpoint and propagating it further is an atomic event). The final snapshot T is defined by resetting all processes that have not taken a checkpoint at S to their starting point. So, in this case $I = \overline{CP(S)}$, the complement set of $CP(S)$.

Theorem 1. *Let π be a distributed computation from S_0 to S with completed checkpointing. Then the final snapshot T obtained by resetting to the initial state all processes that have not taken a checkpoint at S is consistent with π.*

Proof. The final snapshot is defined by $T := r(\overline{CP(S)}, f(CP(S), S))$. The histories of T are given by

$$p_i^T = \begin{cases} p_i^{f(CP(S),S)} & \text{if } P_i \in CP(S), \\ \varepsilon & \text{otherwise.} \end{cases}$$

By Lemma 2 we know that $f(CP(S), S)$ is reachable from S_0 as $CP(S)$ is a freeze set of S. So there is a computation

$$\pi' = S_0 \xrightarrow{e_1} U_1 \xrightarrow{e_2} \dots \xrightarrow{e_m} U_m = f(CP(S), S).$$

Let I stand for $\overline{CP(S)}$. From π' we extract the computation which restricts to events performed by processes not in I, only.

$$S_0 \xrightarrow{\tilde{e}_1} r(I, U_1) \xrightarrow{\tilde{e}_2} \dots \xrightarrow{\tilde{e}_m} r(I, U_m)$$

where $r(I, U_t) \xrightarrow{\tilde{e}_{t+1}} r(I, U_{t+1})$ stands for $r(I, U_t) = r(I, U_{t+1})$ if e_{t+1} is an event of a process in I, and for $r(I, U_t) \xrightarrow{\tilde{e}_{t+1}} r(I, U_{t+1})$ otherwise. For the former case there is nothing to prove, so consider the case e_{t+1} is performed by a process outside I. Again, if e_{t+1} is not of the form $rec_{ij}(msg)$ where $P_i \in I$, $P_j \in CP(S)$, the transition can obviously be performed. for the case $e_{t+1} = rec_{ij}(msg)$, $P_i \in I$, $P_j \in CP(S)$ we show that it cannot occur. So suppose there was such an event, then

$$p_j^{U_{t+1}} = p_j^{U_t}.rec_{ij}(msg)$$

$$p_j^{U_t}.rec_{ij}(msg) \sqsubseteq p_j^{f(CP(S),S)} \qquad \text{by Rule 2.1} \qquad (1)$$

$$p_j^{f(CP(S),S)} \sqsubseteq p_j^S \qquad \text{by Lemma 2} \qquad (2)$$

$$p_j^{f(CP(S),S)} = P_j^S\downarrow_{mcp,cp} \qquad \text{by definition} \qquad (3)$$

This implies $rec_{ij}(\langle msg, 0\rangle)$ is in the history of p_j^S(by (1) and (2)) and occurs before $mcp\_taken_i^S$ or $cp\_taken_i^S$(by (1) and (3)). Hence, by Rule 2.1, $dep_j^S(i) = 1$. Thus, since P_j had taken a checkpoint after the receive event it had also sent a cpr_j to P_i. Now, as $P_i \in I$, this event had not been received and must therefore be in C_{ji}^S. This, however, contradicts that the checkpointing was complete.

	Mutable Checkpointing	Blocking Queue Algorithm
blocking processes	no	no
delaying processing of messages	no	yes
PSN consistent	no	yes
Actions to be taken after phase I	discard unconverted mutable checkpoints	clear blocking queues
Autonomy of processes	no	yes

Table 2. Comparison of the two algorithms.

6 Comparison of Mutable Checkpointing with the Blocking Queue Algorithm

Like mutable checkpointing, the blocking queue algorithm in [13] aims at a reduced coordination overhead. It assumes the same system model. The main difference is that the receipt of flagged messages which would lead to mutable checkpoints are buffered in so-called blocking queues and are not processed until the necessary checkpoints have been taken. This can be viewed as blocking channels or blocking processes partially. Mutable checkpointing neither blocks channels nor processes but, in general, it takes mutable checkpoints which if not converted to checkpoints need to be discarded after completion of the checkpointing (i.e. after phase I). A garbage collection is not required in the blocking queue algorithm, but the blocking queues need to be cleared.

The snapshots determined by the two algorithms over the same underlying computation, in general, are incomparable. That is, it is not the case that a checkpoint taken in mutable computing is always equal or earlier than the corresponding checkpoint in the blocking queue setting, or vice versa. It may even happen that a process takes a checkpoint in one setting but not in the other. We discuss this next. It should be clear that a converted mutable checkpoint can be earlier than the corresponding checkpoint taken by the blocking queue algorithm. A computation in which a process takes a checkpoint in the blocking queue algorithm but not in mutable checkpointing is illustrated in Figure 2. We use M to depict a mutable checkpoint, $\underline{M}$ for a mutable checkpoint converted to a checkpoint, and $\otimes$ for a (regular) checkpoint. Arcs with numbers reflect messages and their flag. Checkpoint requests are indicated by *cpr*. In Figure 2, the checkpoint of P_i in the blocking queue algorithm has no counterpart in mutable checkpointing. Such checkpoints, however, can lead to earlier checkpoints in the blocking queue algorithms and in turn lead to fewer checkpoints than in mutable checkpointing.

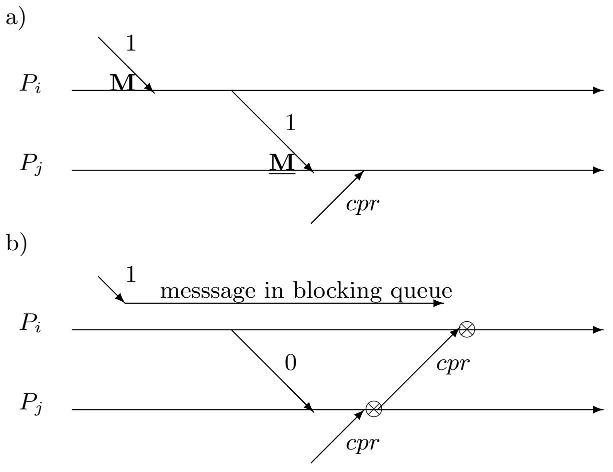

Fig. 2. a) Mutable checkpointing, b) blocking queue algorithm. Process P_i takes a checkpoint in in b), only.

With regard to the potential snapshots, it has been established in [13] that the potential snapshots of the blocking queue algorithm are always consistent, contrasting the general inconsistency of the potential snapshots of mutable checkpointing shown here. In particular, the former implies that one can always only reset those processes for which a checkpoint has been taken without losing consistency. In mutable checkpointing the processes not taking part in the checkpointing also need to be reset as they may have received messages sent out after a checkpoint. Similarly, the garbage collection after phase I does also involve the non-participating processes while for clearing the blocking queues it is sufficient for the participating processes to send a respective clearing message. For mobile computing environments the blocking queue algorithm seems therefore preferable, but it comes together with the temporary buffering and delaying of messages. The comparison is summed up in Table 2. Note, that we did not discuss autonomy of processes here as it is a feature present in [13] independent of the others (but it utilizes the blocking queues). In brief, it allows processes not to take a checkpoint immediately but to postpone it to a time more suitable.

7 Conclusions

We gave a concise specification of the operational behavior of mutable checkpointing, set up an invariant for the reachable states (Lemma 2) and utilized it for the correctness proof of the final snapshot. We extracted the specification of the operational behavior from the pseudo code in [9]. With our translation we omitted a feature not part of the mutable checkpointing concept (the $sent_i$ condition) and in this way obtained a more concise presentation of the algorithm. This feature, however, reduces the number of checkpoints further. It needs to be worked out how it reflects in the potential snapshot.

Taking aside the initial work [10] most papers on checkpointing prove correctness by contradiction. We believe that a direct approach provides more insight into an algorithm. It can also form the basis of a tool-supported proof as recently shown in [3] for Chandy/Lamport's snapshot algorithm (among others). The set-up in [3] is based on the Event-B modelling language [6] and very similar to ours. Whether the proof given in this paper can be mechanized in a similar way is subject of future work.

A concise formal model can be the base of qualitative comparisons which would add to existing quantitative comparisons based on simulations, like [1, 12]. We gave such a comparison with the blocking queue algorithm introduced in [13]. That algorithm is conceptually different – it employs partial buffering of channel contents – but the overall objective is the reduction of coordination overhead as for mutable checkpointing.

We further showed that for a given underlying computation the respective snapshots may be incomparable. The potential snapshots of [13] are always consistent unlike those of mutable checkpointing. Moreover, resetting processes and clearing blocking queues can be done in a localized way (that is involving only processes participating in the checkpointing). Hence, for computing environments in which the economic use of resources is crucial, the blocking queue algorithm seems preferable.

Recently, checkpointing has gained new attention in the area of high performance computing where fault tolerance techniques are essential [5, 11, 16]. As the reduction of the coordination overhead may help to improve the overall performance, the algorithm discussed in this paper may be of interest there.

References

1. A. Agbaria and R. Friedman. Model-based performance evaluation of distributed checkpointing protocols. *Performance Evaluation* 65, 2008.
2. D. Aggarwal and A. Kiehn. Analyzing Mutable Checkpointing via Invariants (full version). Technical Report IIIT Delhi, No IIITD-TR-2015-008, 2015.
3. M.B. Andriamiarina and D. Mery and N.K. Singh. Revisiting snapshot algorithms by refinement-based techniques *Computer Science and Information Systems* 11, 2014.
4. Ö. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems.* ACM Press/Addison-Wesley, 1993.

5. A. Boutellier and P. Lemarinier and G. Krawezik and F. Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. *IEEE Inernational Conference on Cluster Computing (Cluster 2013)*, 2012.
6. D. Cansell and D. M ery. The Event-B modelling method - concepts and case studies. In Bjorner, D., Henson, M. (eds) *Logics of Specification Languages*, Springer, 2008.
7. G. Cao and M. Singhal. On coordinated checkpointing in distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 9(12):1213–1225, 1998.
8. G. Cao and M. Singhal. Mutable checkpoints: a new checkpointing approach for mobile computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 12(2):157–172, 2001.
9. G. Cao and M. Singhal. Checkpointing with mutable checkpoints. *Theoretical Computer Science*, 290(2):1127–1148, 2003.
10. K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.
11. J. Elliot and K. Kharbas and D. Fiala and F. Mueller and K. Ferreira and C. Engelmann. Combining partial redundancy and checkpointing for HPC. *Distributed Computing Systems (ICDCS 2012)*, IEEE, 2012.
12. Q. Jiang and Y. Luo and D. Manivannan An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems. *Journal of Parallel Distributed Computing*, 68, 2008.
13. A. Kiehn, P. Raj, and P. Singh. A causal checkpointing algorithm for mobile computing environments. *Distributed Computing and Networking (ICDCN 2014)*, pages 134–148. LNCS 8314, Springer, 2014.
14. A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, 2008.
15. Lai, Ten H., and Tao H. Yang.: On distributed snapshots. *Information Processing Letters* 25, 1987.
16. I. Ljubuncic and R. Giri and A. Rozenfeld and A. Goldis. Be kind, rewind: checkpoint & restore capability for improving reliability of large-scale semiconductor design. In *IEEE high Performance Extreme Computing Conference (HPEC 2014)*, 2014.
17. M. Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer, 2013.