On the Liveness of Transactional Memory

Victor Bushkov Rachid Guerraoui Michał Kapałka
EPFL, IC, LPD
victor.bushkov@epfl.ch rachid.guerraoui@epfl.ch michal.kapalka@epfl.ch

Abstract

Despite the large amount of work on Transactional Memory (TM), little is known about how much liveness it could provide. This paper presents the first formal treatment of the question. We prove that no TM implementation can ensure local progress, the analogous of wait-freedom in the TM context, and we highlight different ways to circumvent the impossibility.

1. Introduction

Transactional memory (TM) [3–5] is a concurrency control paradigm that aims at simplifying concurrent programming. Each thread of an application performs operations on shared data within a *transaction* and then either commits or aborts the transaction. If the transaction is committed, then the effects of its operations become visible to subsequent transactions; if it is aborted, then the effects are discarded. Transactions are viewed as a simple way to write concurrent programs and hence leverage multicore architectures. Not surprisingly, a large body of work has been dedicated to implementing the paradigm and reducing its overheads.

To a large extent, however, setting the theoretical foundations of the TM concept has been neglected. Indeed, correctness conditions for TMs have been proposed in [1, 2, 18] and programming language level semantics of specific classes of TM implementations have been determined, e.g., in [6–9]. All those efforts, however, focused solely on *safety*, i.e., on what TMs *should not do*. Clearly, a TM that ensures only a safety property can trivially be implemented by permanently blocking all operations. To be meaningful, a TM has to ensure some *liveness* property [10], i.e., a guarantee about what *should be done*.

1.1 Liveness of a TM

In classical shared-memory systems, a liveness property describes when a process that invokes an operation on a shared object is guaranteed to return from this operation [20]. A widely studied such property is *wait-freedom* [11]. It ensures, intuitively, that *every* process invoking an operation eventually returns from this operation, even if other processes crash. It is the ultimate liveness property in concurrent computing as it ensures that every process makes progress.

In a transactional context, requiring such a property alone would however not be enough to ensure any meaningful progress: processes of which all transactions are *aborted* might be satisfying wait-freedom but would not be making real progress. To be meaningful, a TM liveness property should ensure transaction *commitment*, beyond operation *termination*.

One would expect from a TM that every process that keeps executing a transaction (say keeps retrying it in case it aborts) eventually commits it—a property that we call *local progress* and that is similar in spirit to wait-freedom [11]. Not satisfying this property means that some transaction, even when retried forever, might never commit.

In fact, a TM implementation that protects transactions using a single fair global lock could ensure local progress: such a TM would execute all transactions sequentially, thus avoiding transaction conflicts. Yet, such a TM would force processes to wait for each other, preventing them from progressing independently. A process that acquires a global lock and gets suspended for a long time (e.g., due to preemption, page faults, or I/O), or that enters an infinite loop and keeps running forever without releasing the lock, would prevent all other processes from making any progress. This would go against the very essence of wait-freedom. [1] Hence, to be really meaningful a TM liveness property should enforce some "independent" progress.

1.2 Transaction Failures

The classical way of modeling shared-memory systems in which processes can make progress independently, i.e., without waiting for each other, is to consider *asynchronous* systems in which processes can be arbitrarily slow, including failing by *crashing*. (This typically models behaviors such as paging to disk.) A TM implementation that is resilient to crashes enables the progress of a process even if other processes are suspended for a long time. In the same vein, one should also ensure progress in the face of *parasitic* processes—those that keep executing transactional operations without ever attempting to commit. These model long-running processes whose duration cannot be anticipated by the system, e.g., because of an infinite loop.

To illustrate the underlying challenges, consider the following example, depicted in Figure 1. Two processes, p_1 and p_2, execute transactions T_1 and T_2, respectively. Process p_1 reads value 0 from a shared variable x and then gets suspended for a long time. Then, process p_2 also reads value 0 from x, and attempts to write value 1 to x and commit. Because of asynchrony, the processes can be arbitrarily delayed. Hence, the TM does not know whether p_1 has crashed or is just very slow, and so, in order to ensure the progress of process p_2, the TM might eventually allow process p_2 to commit

[1] Amdahl's Law, recommending to reduce the sequential parts of the programs indirectly stipulates that transactions should not wait for each other. Resilient TM implementations, which allow transactions to make progress independently, may provide better performance on future hardware with a big number of computing cores. Resilient (lock-free) implementations of concurrent data structures, which could be considered inefficient when compared to lock-based ones on single-processor systems, have already entered the mainstream because of their better scalability on modern hardware.

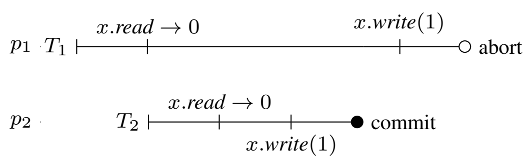

Figure 1. An illustration of the difficulty of ensuring local progress. The scenario can repeat infinitely many times preventing transaction T_1 from ever committing.

transaction T_2. But then, if process p_1 writes value 1 to x and attempts to commit transaction T_1, the TM cannot allow process p_1 to commit, as this would violate safety. A similar situation can occur in the case of parasitic processes, say if p_1 keeps repeatedly reading from variable x. If the maximum length of a transaction is not known, the TM cannot say whether p_1 is parasitic or not, and thus may eventually allow process p_2 to commit transaction T_2, forcing process p_1 to abort T_1 later.

1.3 Contributions

This paper first introduces the notion of a *TM-liveness* property which specifies for each infinite executions which processes should make progress, i.e. which processes commit transactions infinitely often. We formalize this notion by modeling TM implementations as I/O automata and focusing on infinite histories of such automata.

We prove an impossibility result which states that no TM implementation can ensure both *local progress* and *opacity* in a fault-prone system, i.e. in a system in which any number of processes can crash or be parasitic. Opacity is the safety property ensured by most TM implementations. It states that every transaction observes a consistent state of the system. Local progress is a TM-liveness property, highlighted above, which states that every correct process, i.e. a process which is not parasitic and does not crash, makes progress. We actually prove a more general result stating that no TM implementation can ensure, besides any safety property that is at least as strong as serializability, the progress (I) of at least two correct processes in the system, as well as of (II) any correct process that runs alone.

There are at least two ways to circumvent our impossibility result. One way is to weaken local progress and require only *global progress*, a TM-liveness property that guarantees progress for at least one correct process. There are implementations that ensure opacity and global progress, e.g., OSTM [13]. We present a simple TM implementation that ensures opacity and global progress. We give a simple formal description of it and prove it is correct. A second way is to assume that a TM entirely controls the user application, i.e. that it can re-execute portions of a program that uses the TM or simulate an execution of a given transaction [12].

2. Preliminaries

2.1 Processes and Shared Memory

We assume a classical (see, e.g., [11]) asynchronous, shared memory system of n *processes* $p_1, \ldots, p_n$ that communicate by executing operations on *base objects* (which represent the shared memory, e.g., provided in hardware). A *shared object* is a higher-level abstraction provided to processes, and implemented typically in software using base objects.

For instance, if base objects are memory locations with basic operations such as read, write, and compare-and-swap, then shared objects could be shared data structures such as linked lists or hash tables. If a process p_i invokes an operation *op* on a shared object O, then p_i follows the implementation of O, possibly accessing any number of base objects and executing local computations, until p_i is returned a result of *op*. We assume that processes are sequential; that is, whenever a process p_i invokes an operation *op* on any shared object, p_i does not invoke another operation on any shared object until p_i returns from *op*. Invocations and responses of shared objects operations are called (invocation and response) *events*.

2.2 Transactional Memory

Let K be a set of *process identifiers*, $P = \{p_k | k \in K\}$ be a set of processes, and X be a set of base objects called *t-variables* ("t-variable" stands for "transactional variable"). For presentation simplicity, we focus on t-variables that support wait-free *read* and *write* operations. Each t-variable can have values from set V. Let $Inv_k = \{x.write^k(v) | x \in X \text{ and } v \in V\} \cup \{x.read^k | x \in X\} \cup \{tryC^k\}$ be the set of invocation events of process p_k and $Res_k = \{v^k | v \in V\} \cup \{ok^k, A^k, C^k\}$ be the set of response events of process p_k, C^k is a *commit event* and A^k is an *abort event*. Also, let $Inv = \cup_{k \in K} Inv_k$ and $Res = \cup_{k \in K} Res_k$.

Since a TM is a discrete event system that receives inputs from processes and returns corresponding responses we model the behavior of TM implementations using I/O automata. Formally, an I/O automaton F is a quintuple (St, I, O, s_0, R), where:

- St is a (possibly infinite) set of states.
- I is a set of input events.
- O is a set of output events.
- s_0 is the initial state.
- $R \subseteq St \times (I \cup O) \times St$ is a transition relation.

The sets I and O are disjoint. A *history* H of automaton $F = (St, I, O, s_0, T)$ is a (finite or infinite) sequence of events $H = e_1 \cdot e_2 \cdot \ldots$ over alphabet $(I \cup O)$ for which there exists a finite or infinite sequence of states $s_0 \cdot s_1 \cdot \ldots$, where each $s_i \in St$ and s_0 is the initial state, such that for any $i \in \{1, 2, \ldots\}$ we have $(s_{i-1}, e_i, s_i) \in T$. The set of all histories of F is denoted by $H(F)$.

Let Σ_k be an alphabet elements of which are words derived by concatenating invocation and response events associated with process p_k such that $\Sigma_k = \{x.write^k(v) \cdot ok^k | x.write^k(v) \in Inv_k\} \cup \{x.read^k() \cdot v^k | x.read^k() \in Inv_k \text{ and } v^k \in Res_k\} \cup \{tryC^k \cdot C^k\} \cup \{e \cdot A^k | e \in Inv_k\}$. Also, let Σ_k^∞ be a set of all finite and infinite sequences over alphabet Σ_k. Let H be a history over alphabet $Inv \cup Res$, we define a *projection* $H|p_k$ of H on process p_k as the longest subsequence of H consisting of events from $Inv_k \cup Res_k$.

A *TM implementation* M is an I/O automaton $F = (St, I, O, s_0, R)$ such that $I = Inv$, $O = Res$ and for every history H of F and every $p_k \in P$ we have $H|p_k \in \Sigma_k^\infty$.

Given projection $H|p_k$ of history H of some TM implementation, a *transaction* of p_k in H is a subsequence $T = e_1 \cdot \ldots \cdot e_n$ of $H|p_k$ such that:

- e_1 is the first event in $H|p_k$ or the previous event e_0 in $H|p_k$ is either A^k or C^k, and
- e_n is either A^k or C^k or the last event in $H|p_k$, and
- No event in T, except e_n, is A^k or C^k.

Transaction T is *committed* (*aborted*) if the last event in T is a commit (abort) event. Given transactions T_1 and T_2 in history H, we say that T_1 *precedes* T_2 in H, denoted by $T_1 <_H T_2$, if T_1 is committing or aborting and the last event of T_1 occurs in H before the first event of T_2; otherwise T_1 is *concurrent* to T_2. History H is *sequential* if no two transactions in H are concurrent to each other.

Processes communicate with each other only through a TM implementation M by invoking concurrently requests (read, write,

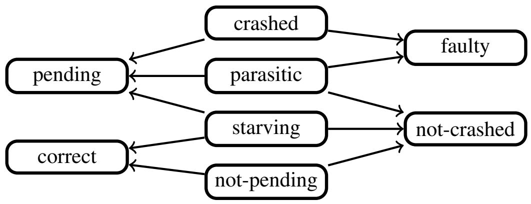

Figure 2. Classes of processes. An arrow from class c_1 to class c_2 means that every process which belongs to c_1 also belongs to c_2.

and commit requests) and receiving corresponding responses from M within transactions. All events within a transaction appear to other transactions as if they occur instantaneously. If a transaction is committed, then all the changes made by write operations within the transaction are made visible to other transactions; otherwise all the changes are rolled back. Processes send commit requests $tryC^k$ to the TM implementation that decides which transactions should be committed or aborted. To reduce contention between transactions, a TM implementation may use a logically separate module called a contention manager. A contention manager can delay or force TM to abort some of the transactions. In this work we consider a contention manager as an integral part of a TM implementation. That is, all the results of the paper apply to the entire TM, including the contention manager.

The order in which processes invoke events is determined by a *scheduler*. Processes and TM implementations have no control over a scheduler. At any point in time the scheduler decides invocation event of which process is going to be given to the TM implementation. These decisions form a *schedule* which is a finite or an infinite sequence of process identifiers.

2.3 Process Failures

We say that process p_k is *pending* in infinite history H if H has only a finite number of commit events C^k. Process p_k *crashes* in infinite history H if $H|p_k$ is a finite non-empty sequence. That is, from some point in time p_k does not execute any events.

Intuitively, a *parasitic* process is a process that keeps executing operations but, from some point in time, never attempts to commit (by invoking operation $tryC$) when given a chance to do so. Consider any infinite history H, and process p_k in H. If process p_k from some point in time executes infinitely many operations without being aborted and without attempting to commit, then p_k is parasitic. On the contrary, if p_k invokes operation $tryC^k$ or is aborted infinitely many times, then p_k is not parasitic. Formally, we say that process p_k is parasitic in infinite history H if $H|p_k$ is infinite and in history $H|p_k$ there are only a finite number of invocations $tryC^k$ and abort events A^k. If a process does not crash, is not parasitic, and is pending in infinite history H, then it is *starving* in H.

We say that process p_k is *correct* in infinite history H if p_k is not parasitic in H and does not crash in H. If a process is not correct in H, then it is *faulty* in H. Figure 2 depicts the relations between different classes of processes.

We define a *fault-prone system* *Sys* to be any of the following:

- a system in which any number of processes can crash or be parasitic, or
- a system in which any number of processes can crash, but no process is parasitic (we call such system *parasitic-free*), or
- a system in which any number of processes can be parasitic, but no process crashes (we call such system *crash-free*).

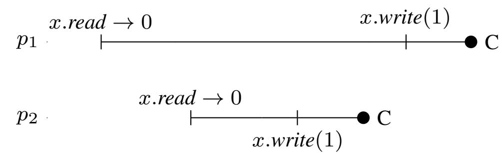

Figure 3. A history which is not opaque and not strictly serializable. Process identifiers are omitted for simplicity.

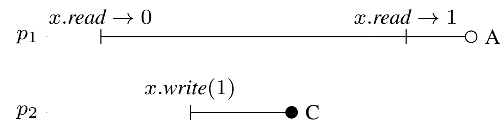

Figure 4. A history which is strictly serializable but not opaque.

2.4 Safety properties of TM

Intuitively a safety property S states that some events should never happen. We consider two safety properties of TM implementations: strict serializability S_s and opacity S_o. Intuitively, strict serializability requires every committed transaction to observe a consistent state of the system [19], while opacity requires every transaction (even aborted) to observe a consistent state of the system [18].

We say that history H is *equivalent* to history H' if for every process $p_k \in P$ we have $H|p_k = H'|p_k$. We obtain the *completion* $com(H)$ of finite history H by aborting every transaction which is neither committed nor aborted, i.e. by adding to the end of the history corresponding abort events. If $com(H) = H$, then history H is *complete*. We say that finite history H' *preserves the real time order* of finite history H if for any two transactions T_1 and T_2 in H if $T_1 <_H T_2$, then $T_1 <_{H'} T_2$. Let H_s be a complete sequential history and T_j be a transaction in H. Denote by $visible(T_j)$ the longest subsequence of H_s such that for every transaction T'_j in the subsequence, either $j' = j$ or $T'_j <_{H_s} T_j$. Transaction T_j is *legal* in H_s if for every t-variable $x \in X$ history $visible(T_j)$ respects the semantics of x, i.e. for every transaction T'_j in $visible(T_j)$ and every response event v^k, $k \in K$, v is the value of the previous write to x invocation event in T'_j or v is the value of x when T'_j starts if there are no write to x invocation events in T'_j before v^k.

A finite history H is *strictly serializable* if there exists a sequential history H_s equivalent to H_{com}, where H_{com} is the longest subsequence of H containing only committed transactions, such that H_s preserves the real-time order of H, and every transaction in H_s is legal. A finite history H is *opaque* if there exists a sequential history H_s equivalent to $com(H)$, such that H_s preserves the real-time order of $com(H)$, and every transaction in H_s is legal. Let M be a TM implementation represented by I/O automaton F. We say that M ensures strict serializability (respectively opacity) iff every finite history H of F is strictly serializable (respectively opaque).

For example, the history in Figure 1 is opaque, while the histories in Figure 3 and Figure 4 are not opaque. The histories in Figure 1 and Figure 4 are strictly serializable, while the history in Figure 3 is not strictly serializable.

3. Liveness of a TM

We introduce in this section the concept of a *TM-liveness* property and we give examples of such properties.

3.1 TM-liveness Properties

Basically, a TM-liveness property states whether some process p_k should make progress in some infinite history H. Clearly, progress

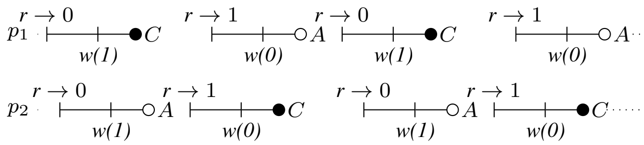

Figure 5. An infinite history with two processes and one t-variable. Each process executes an infinite number of transactions which read value 0 (read value 1) and write value 1 (write value 0). For simplicity process identifiers in the operations are omitted, $r \to v$ means that a process reads value v from the t-variable and $w(v)$ means that a process writes value v to the t-variable.

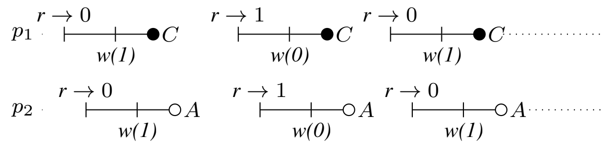

Figure 6. An infinite history with two processes and one t-variable. Processes execute an infinite number of transactions which read value 0 (read value 1) and write value 1 (write value 0).

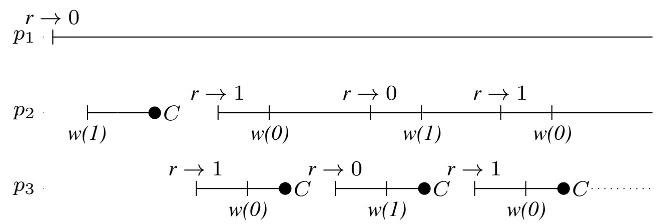

Figure 7. An infinite history with three processes and one t-variable. Process p_1 starts a transaction by invoking a read operations, but then it crashes. Process p_2 executes two transactions, but it becomes parasitic in the second transaction. Process p_3 executes an infinite number of transactions which read value 0 (read value 1) and write value 1 (write value 0).

cannot be required for crashed or parasitic processes: these processes have executions with a finite number of $tryC$ operation invocations. We define a TM-liveness property as a weakening of the strongest TM-liveness property. The strongest TM-liveness property guarantees that in every infinite history of a TM implementation every correct process makes progress.

Formally, a correct process p_k in infinite history H *makes progress* in H iff p_k is not pending H. Let H_{TM} be the set of all infinite histories H for which there exists a TM implementation represented by automaton F such that $H \in H(F)$.

We define *local progress*, which is analogous to wait-freedom in shared memory, as a set L_{local} of infinite histories from H_{TM} such that infinite history $H \in H_{TM}$ belongs to L_{local} iff the following holds:

- Every correct process in H makes progress in H, or
- H does not have any correct process.

Definition 1. *A* TM-liveness *property L is a set of infinite histories such that $L_{local} \subseteq L \subseteq H_{TM}$.*

Definition 2. *An infinite history H ensures TM-liveness property L iff $H \in L$.*

Let M be a TM shared object represented by I/O automaton F.

Definition 3. *A TM shared object M ensures TM-liveness property L iff every infinite history $H \in H(F)$ ensures L.*

3.2 Examples of TM-liveness Properties

3.2.1 Local Progress

A TM shared object M ensures local progress if M guarantees that every correct process makes progress.

For example, Figure 5 shows an infinite history which ensures local progress in a system with two processes and one t-variable. Both processes make progress (are not pending) in the history.

As we prove in this paper, implementing a TM that ensures opacity and local progress in any fault-prone system is impossible. That is, local progress inherently requires some form of indefinite blocking of transactions. Ensuring local progress in a system that is both crash-free and parasitic-free is possible. It suffices to use a simple TM that synchronizes all transactions using a single global lock, and thus never aborts any transaction.

3.2.2 Global Progress

A TM shared object M ensures *global progress* if M guarantees that some correct process makes progress. We define global progress, as a TM-liveness property L_{global} such that infinite history $H \in H_{TM}$ belongs to L_{global} iff the following holds:

- At least one correct process in H makes progress in H, or
- H does not have correct processes.

Figure 6 depicts an infinite history which ensures global progress in a system two processes and one t-variable. Both of the processes are correct in the history. However, only process p_1 makes progress in the history.

3.2.3 Solo Progress

A TM shared object M ensures *solo progress* if M guarantees that every correct process which eventually runs alone makes progress. A process *runs alone* if starting from some point in time it is concurrent only to processes which are faulty.

Formally, a process p_k runs alone in infinite history H iff p_k is correct in H and no other process is correct in H. We define solo progress, as a TM-liveness property L_{solo} such that infinite history $H \in H_{TM}$ belongs to L_{solo} iff the following holds:

- A process that runs alone in H makes progress in H, or
- H does not have a process that runs alone in H.

Figure 7 depicts an infinite history H_{solo} which ensures solo progress in a system with three processes and one t-variable. Process p_1 crashes, p_2 is parasitic, and p_3 runs alone and makes progress (is not pending).

Obstruction-free TM implementations [14, 18] ensure solo progress in parasitic-free systems. Lock-based TM implementations, such as TinySTM [17] and SwissTM [16], ensure solo progress in systems that are both parasitic-free and crash-free. Those lock-based TMs that use deferred updates, however, such as TL2 [15], ensure solo progress in crash-free systems.

4. Impossibility of Local Progress

Like in any distributed problem, each history of a TM implementation can be thought of as a game between the environment and the implementation. The *environment* consisting of processes and a scheduler decides on inputs (operation invocations) given to the implementation and the implementation decides on outputs (responses) returned to the environment. To prove that there is no TM implementation that ensures both opacity and local progress in a fault prone system we use the environment as an adversary that acts

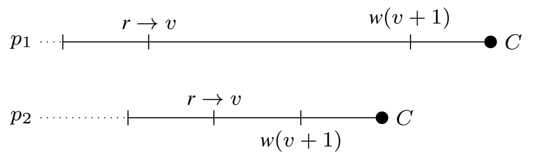

Figure 8. A suffix of a finite history corresponding to an execution of Algorithm 1 when it terminates. For simplicity, $r \to v$ means that a process reads value v from x and $w(v+1)$ means that a process writes value $v+1$ to x, process identifiers are omitted.

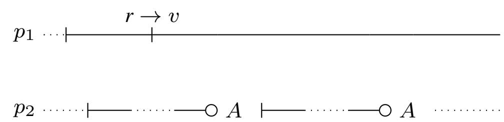

Figure 9. A suffix of an infinite history corresponding to an execution of Algorithm 1 when process p_1 crashes.

against the implementation. The environment wins if the resulting infinite history violates local progress.

Theorem 1. *No TM shared object ensures both local progress and opacity in any fault-prone system.*

Proof. Assume otherwise, i.e. that there exists a TM shared object M represented by I/O automaton F that ensures local progress and opacity in any fault-prone system. To find a contradiction, we exhibit a winning strategy (Algorithms 1 and 2 below) for the environment resulting in an infinite history of F which does not ensure local progress.

By definition, a fault-prone system *Sys* is either a system, in which any number of processes can crash or be parasitic, a crash-free system with a parasitic processes, or a parasitic-free system with crashes. We thus consider three different cases:

***Sys* is parasitic-free.** We exhibit a history which violates local progress even when no process is parasitic.

Consider two processes p_1 and p_2 and the environment that interacts with M using the strategy defined by the following algorithm:

Algorithm 1.

1. **Step 1.** Process p_1 invokes a read operation on t-variable x and receives as a response v^1 or A^1. The algorithm goes to Step 2.

2. **Step 2.** Process p_2 invokes a read operation on t-variable x and receives as a response v^2 or A^2. If M returns A^2, then the algorithm repeats Step 2, otherwise p_2 invokes an operation on x, which writes value $v+1$ to x, and receives as a response ok^2 or A^2. If the response is A^2, then the algorithm repeats Step 2, otherwise p_2 invokes $tryC^2$ operation and receives a response from M. If M returns C^2, then the algorithm goes to Step 3, otherwise it repeats Step 2.

3. **Step 3.** If the last response that p_1 received at Step 1 is A^1, then the algorithm goes to Step 1. Otherwise, process p_1 invokes a write operation on t-variable x which writes value $v+1$ to x, and then receives a response from M. If the response is A^1, then the algorithm goes to Step 1, otherwise p_1 invokes $tryC^1$ operation and receives a response from M. If M returns C^1, then the algorithm stops, otherwise the algorithm goes to Step 1.

We first show that there exists an infinite history of M corresponding to an execution of Algorithm 1. To do so, we prove that Algorithm 1 never terminates, i.e. that at Step 3 process p_1 is never returned C^1 by M in any history of M corresponding to an execution of the algorithm. Assume some finite history H_f of F corresponding to an execution of Algorithm 1 such that the last event in H_f is C^1. A suffix of history H_f is shown in Figure 8.

Since M ensures opacity, there exists a sequential finite history H_s which is equivalent to $com(H_f)$, preserves the real-time order of $com(H_f)$, and every transaction in H_s is legal. Since history H_f has no transactions which are neither committed nor aborted, then $com(H_f) = H_f$. Hence H_s is equivalent to H_f and preserves the real-time order of H_f. Since H_s is a sequential history and preserves the real-time order of H_f, then H_s could only have one of the following forms, where H'_s is a prefix of H_s:

1. $H_s = H'_s \cdot x.read^1() \cdot v^1 \cdot x.write^1(v+1) \cdot ok^1 \cdot tryC^1 \cdot C^1 \cdot x.read^2() \cdot v^2 \cdot x.write^2(v+1) \cdot ok^2 \cdot tryC^2 \cdot C^2$
2. $H_s = H'_s \cdot x.read^2() \cdot v^2 \cdot x.write^2(v+1) \cdot ok^2 \cdot tryC^2 \cdot C^2 \cdot x.read^1() \cdot v^1 \cdot x.write^1(v+1) \cdot ok^1 \cdot tryC^1 \cdot C^1$.

In the first case, the last transaction executed by process p_2 is not legal in H_s, because p_2 reads value v from t-variable x the value of which is $v+1$ and this violates the semantics of x. In the second case, the last transaction executed by process p_1 is not legal in H_s, because p_1 reads value v from t-variable x the value of which is $v+1$, this leads to violation of the specification of x. Thus, H_f is not opaque. Since every history H_f of M that ends with commit event C^1 is not opaque and M ensures opacity, then H_f is not a history of M corresponding to the execution of the algorithm. In other words, every history of M corresponding to the execution of Algorithm 1 is infinite.

Consider some infinite history H of M corresponding to the execution of the above algorithm. Since process p_1 never receives commit event C^1 from F, then p_1 is pending in H. Since *Sys* is parasitic-free, then process p_1 can crash in history H. Therefore, we focus on the following two cases:

- **Process p_1 crashes in history H.** A suffix of such history H is depicted in Figure 9. According to the algorithm, process p_1 can crash in infinite history H iff process p_2 is pending and invokes infinitely many operations. Process p_2 can invoke infinitely many operations iff the algorithm executes infinitely many iterations of Step 2. At each iteration of Step 2 process p_2 invokes operation $tryC^2$, thus p_2 is correct in H. Since M ensures local progress and p_2 is correct in H, then process p_2 is not pending: a contradiction. Thus, H does not ensure local progress.

- **Process p_1 does not crash in history H.** A suffix of such history H is depicted in Figure 10. Since H is infinite and p_1 does not crash in H, then according to the algorithm p_1 invokes infinitely many operations and receives infinitely many abort events at Step 3. Thus, p_1 is a correct process in H. Since M ensures local progress, then p_1 makes progress in H, which means that eventually p_1 is returned commit event $C^1$and history H is not infinite: a contradiction. Thus, H does not ensure local progress.

Thus, in a parasitic-free system *Sys*, TM object M cannot ensure both local progress and opacity.

***Sys* is crash-free.** We exhibit a history which violates local progress even when no process crashes.

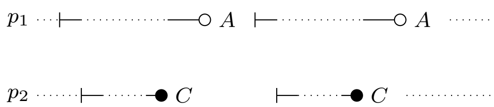

Figure 10. A suffix of an infinite history corresponding to an execution of Algorithm 1 when process p_1 does not crash.

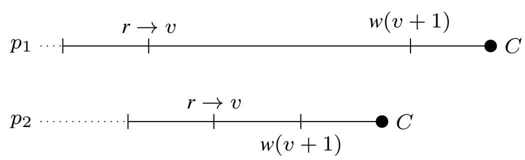

Figure 11. A suffix of a finite history corresponding to an execution of Algorithm 2 when it terminates.

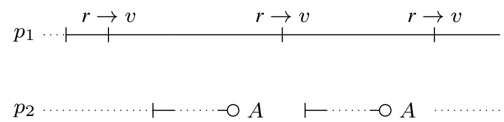

Figure 12. A suffix of an infinite history corresponding to an execution of Algorithm 2 when process p_1 is parasitic.

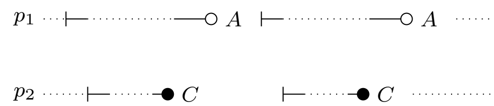

Figure 13. A suffix of an infinite history corresponding to an execution of Algorithm 2 when process p_1 is not parasitic.

Consider two processes p_1 and p_2 and the environment that interacts with M using the strategy defined by the following algorithm:

Algorithm 2.

1. **Step 1.** Process p_1 invokes a read operation on t-variable x and receives as a response v^1 or A^1. Process p_2 invokes a read operation on t-variable x and receives as a response v^2 or A^2. If the response is A^2, then the algorithm repeats Step 1, otherwise p_2 invokes a write operation which writes value $v+1$ to x, and then p_2 receives a response ok^2 or A^2. If the response is A^2, then the algorithm repeats Step 1, otherwise p_2 invokes $tryC^2$ operation and receives a response from M. If M returns C^2, then the algorithm goes to Step 2, otherwise it repeats Step 1.

2. **Step 2.** If the last response that p_1 received from M is A^1, then the algorithm goes to Step 1. Otherwise, process p_1 invokes a write operation on t-variable x which writes value $v+1$ to x, and then p_1 receives a response from M. If the response is A^1, then the algorithm goes to Step 1, otherwise p_1 invokes $tryC^1$ operation and receives a response from M. If M returns C^1, then the algorithm stops, otherwise the algorithm goes to Step 1.

First, we prove that Algorithm 2 never terminates, i.e. that at Step 2 process p_1 is never returned C^1 by M in any history of M corresponding to an execution of the algorithm. Assume some finite history H_f of F corresponding to an execution of Algorithm 2 such that the last event in H_f is C^1. A suffix of history H_f is shown in Figure 11.

Since M ensures opacity, there exists a sequential finite history H_s which is equivalent to $com(H_f)$, preserves the real-time order of $com(H_f)$, and every transaction in H_s is legal. Since history H_f has no transaction which are neither committed nor aborted, then $com(H_f) = H_f$. Hence H_s is equivalent to H_f and preserves the real-time order of H_f. Since H_s is a sequential history and preserves the real-time order of H_f, then H_s could only have one of the following forms, where H'_s is a prefix of H_s:

1. $H_s = H'_s \cdot x.read^1() \cdot v^1 \cdot x.write^1(v+1) \cdot ok^1 \cdot tryC^1 \cdot C^1 \cdot x.read^2() \cdot v^2 \cdot x.write^2(v+1) \cdot ok^2 \cdot tryC^2 \cdot C^2$
2. $H_s = H'_s \cdot x.read^2() \cdot v^2 \cdot x.write^2(v+1) \cdot ok^2 \cdot tryC^2 \cdot C^2 \cdot x.read^1() \cdot v^1 \cdot x.write^1(v+1) \cdot ok^1 \cdot tryC^1 \cdot C^1$.

In the first case, the last transaction executed by process p_2 is not legal in H_s, because p_2 reads value v from t-variable x the value of which is $v+1$ and this violates the semantics of x. In the second case, the last transaction executed by process p_1 is not legal in H_s, because p_1 reads value v from t-variable x the value of which is $v+1$, this leads to violation of the specification of x. Thus, H_f is not opaque. Since every history H_f of M that ends with commit event C^1 is not opaque and M ensures opacity, then H_f is not a history of M corresponding to the execution of the algorithm. In other words, every history of M corresponding to the execution of Algorithm 2 is infinite.

Consider now some infinite history H of M corresponding to the execution of the above algorithm. Since process p_1 never receives commit event C^1 from F, then p_1 is pending in H. Since S is crash-free, then process p_1 can be parasitic in history H. Therefore, we focus on the following two cases:

- **Process p_1 is parasitic in history H.** A suffix of such history H is shown in Figure 12. According to the algorithm, process p_1 can be parasitic in infinite history H iff process p_2 is pending and invokes infinitely many operations at Step 1 without receiving a commit event C^2. Process p_2 can invoke infinitely many operations iff the algorithm executes infinitely many iterations of Step 1. At each iteration of Step 1 process p_2 either receives abort event A^k or invokes operation $tryC^2$, thus p_2 is correct in H. Since M ensures local progress, then p_2 makes progress in H, i.e. process p_2 is not pending: a contradiction. Thus, H does not ensure local progress.

- **Process p_1 is not parasitic in history H.** A suffix of such history H is shown in Figure 13. According to Algorithm 2 H can be infinite iff the algorithm executes Step 1 infinitely often, the algorithm executes Step 1 infinitely often iff process p_1 invokes infinitely many operations. Since p_1 invokes infinitely many operations and p_1 is pending in H, then p_1 receives infinitely many abort events in H. Thus, history p_1 is correct in H. Since M ensures local progress, then p_1 makes progress in H, which means that eventually p_1 is returned commit event C^k and H is finite: a contradiction. Thus, H does not ensure local progress.

Hence, we proved that in a crash-free system *Sys* TM object M cannot ensure both local progress and opacity.

Sys is not crash-free or parasitic free. Since in *Sys* any number of processes can crash or be parasitic, there are no restrictions on the inputs provided by the environment. Thus, we can use Algorithm 1 (or Algorithm 2) to exhibit an infinite history that does not ensure local progress. □

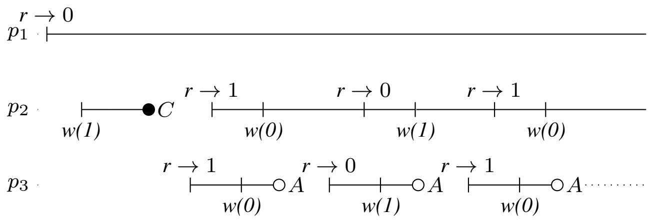

Figure 14. An infinite history with three processes and one t-variable. Process p_1 starts a transaction by invoking a read operations, but then it crashes. Process p_2 executes two transactions, but it becomes parasitic in the second transaction. Process p_3 executes an infinite number of aborting transactions which read value 0 (read value 1) and write value 1 (write value 0).

5. Generalizing the Impossibility

We generalize here the result of the previous section; namely, we determine a larger class of TM-liveness properties and a larger class of safety properties that are impossible to implement in a fault-prone system. In short, we show that a TM cannot ensure the progress of at least two correct processes as well as the progress of any process that runs alone.

5.1 Classes of properties

Nonblocking TM-liveness properties. Intuitively, we say that a TM-liveness property is *nonblocking* if it guarantees progress for every correct process that eventually runs alone. More precisely:

Definition 4. *A TM-liveness property L is nonblocking iff for every $H \in L$ if some process runs alone in H, then the process makes progress in H.*

For example, Figure 5, Figure 6, and Figure 7 show infinite histories which ensure nonblocking TM-liveness properties while Figure 14 shows an infinite history which does not ensure any nonblocking TM-liveness property. TM-liveness properties that are not nonblocking are called *blocking*. Local progress and solo progress are nonblocking. Note that solo progress is weaker than every nonblocking property while local progress is the strongest among nonblocking properties.

Biprogressing TM-liveness properties. Intuitively, we say that a TM-liveness property L is a *biprogressing* property if for every infinite history it guarantees that at least two correct processes make progress. More precisely:

Definition 5. A *TM-liveness property* $L = \{L^1, \ldots, L^n\}$ *is* biprogressing *iff for every $H \in L$ if at least two processes are correct in H, then at least two processes make progress in H.*

For example, Figure 5 and Figure 7 show infinite histories which ensure a biprogressing property while Figure 6 shows an infinite history which does not ensure any biprogressing property. Local progress is a biprogressing property while global progress and solo progress are not biprogressing.

Strictly serializable safety properties. We say that that a safety property S is strictly serializable if it is stronger (or equal) then strict serializability S_s. Formally, a safety property S is *strictly serializable* iff for every TM implementation M if M ensures S, then M ensures strict serializability. Both strict serializability and opacity are strictly serializable safety properties.

5.2 Generalized Result

We show that TM-liveness properties that are nonblocking and biprogressing are impossible to implement together with a strictly serializable safety property in any fault-prone system. We start by stating the following lemma, which says, intuitively, that a process executing infinitely many transactions can block the progress of all other processes if the TM ensures any nonblocking TM-liveness property. The proof of the lemma follows the same line of reasoning as in Theorem 1. The main difference is that to prove the lemma we need to exhibit a history in which process executing infinitely many transactions blocks the progress of other processes for an arbitrary number of processes n, while in Theorem 1 we exhibit such history for two processes.

Lemma 1. *For every TM implementation represented by I/O automaton F that ensures a strictly serializable safety property and a nonblocking TM-liveness property in any fault-prone system, there exists an infinite history H of F such that at least two processes are correct in H and at most one process makes progress in H.*

Proof. Let M be a TM implementation ensuring a nonblocking TM-liveness property in a fault-prone system *Sys* and F be its I/O automaton representation. To exhibit a history in which at least two processes are correct and at most one process makes progress we consider a game between the environment and the implementation. The environment acts against the implementation and wins the game if the resulting history satisfies the requirements above.

By definition, a fault-prone system *Sys* is either a system in which any number of processes may crash or be parasitic, a crash-free system, or a parasitic-free system. We thus consider three different cases:

Sys is parasitic-free. Consider two processes p_1 and p_2 that interact with M. The strategy that the environment uses to win the game is described by the Algorithm 1. We proved in Theorem 1 that there is no history that corresponds to a terminating execution of the algorithm. The algorithm never terminates and all histories corresponding to an execution of the algorithm are infinite.

Consider some infinite history H corresponding to an execution of the algorithm. Since *Sys* is parasitic-free, process p_1 either crashes in history H or does not crash in H. Assume that process p_1 crashes in history H. According to the algorithm, process p_1 can crash in infinite history H only if process p_2 is pending and invokes infinitely many operations, i.e. only if p_2 is returned an infinite number of abort events at Step 2. Since p_2 is returned an infinite number of abort events, p_n is correct in H . Because after some time only process p_2 executes operations in H (i.e. p_2 runs alone in H) and M ensures a TM-liveness property which is nonblocking, then p_2 makes progress in H, i.e. process p_n is not pending: a contradiction. Thus, p_1 cannot crash in H. According to the algorithm, p_2 cannot crash in H since Step 2 is repeated infinitely often; p_1 and p_2 cannot be parasitic since p_2 is always eventually returned C^2 and p_1 is returned A^1 at Step 1 or Step 3. Thus, in history H both of the processes are correct and at most one process makes progress (since p_1 is never returned C^1).

Sys is crash-free. Consider two processes p_1 and p_2 that interact with M. The strategy that the environment uses to win the game is described by the Algorithm 2. We proved in Theorem 1 that there is no history that corresponds to a terminating execution of the algorithm. The algorithm never terminates and all histories corresponding to an execution of the algorithm are infinite.

Consider some infinite history H corresponding to an execution of the algorithm. Since *Sys* is crash-free, process p_1 is either parasitic or not in H. Assume that p_1 is parasitic in H. According to the algorithm, p_1 can be parasitic only if p_2 is pending in H and returned A^2 infinitely often (i.e. correct). Since a correct process p_2 runs alone in H and M ensures a nonblocking TM-liveness property, then p_2 makes progress in H: a contradiction. Thus, p_1

cannot be parasitic in H. According to the algorithm, p_2 cannot be parasitic in H since p_2 either invokes $tryC^2$ or is returned A^2 infinitely often at Step 1; p_1 and p_2 cannot crash in H since the algorithm repeats Step 1 infinitely often.

Sys is not crash-free or parasitic free. Since in *Sys* any number of processes can crash or be parasitic there are no restrictions on the inputs provided by the environment. Thus, we can use Algorithm 1 (or Algorithm 2) to exhibit an infinite history that does not ensure local progress.

□

By definition, a biprogressing TM-liveness property should ensure progress for at least two correct processes in every infinite history. While, by the above lemma, if the property is also nonblocking, then we can find an infinite history of any TM implementation in a fault prone system in at least two processes are correct and at most one process makes progress: a contradiction. Thus, we end up with the following theorem.

Theorem 2. *For every TM-liveness property L which is nonblocking and biprogressing there is no TM implementation that ensures serializable safety property and L in any fault-prone system.*

6. Ensuring Global Progress

In this section we show that there exists a TM implementation that ensures both opacity and global progress in a fault-prone system. We thus show that every TM-liveness property which is weaker than global progress can be ensured in any fault-prone system. The purpose of the TM implementation given in this section is only to formally prove the possibility of global progress in any fault-prone system—the TM is not meant to be practical or efficient. Note that there are TM implementations that ensure opacity and global progress, e.g., OSTM [13]. However it is not know to us if it has been formally proven.

We build an automaton F_{gp} of the implementation that ensures global progress and opacity in any fault prone system. The main idea of the automaton is the following. A process in a group of concurrent processes, which invokes a commit request first, receives a commit event, and after the process receives it other processes from the concurrent group can receive only an abort event. At each state we keep the set of processes which are concurrent to each other and can receive a commit event at this state. The automaton never returns an abort event to processes before some process from the set of concurrent processes invokes a commit request for which the automaton returns a commit event.

After the automaton sends to some process p_k a commit event, all other processes that were in the concurrent set at the time when the commit event was sent receive only abort events for every operation invocation and are removed from the set of concurrent processes. The process which receives a commit event is also removed from the set of concurrent processes. Then a new concurrent group is formed—every process that starts a new transaction is added to the group. Each state s of the automaton F_{gp} is a tuple $s = (Status, CP, Val, f)$. Where:

- Array $Status$ is a status array which specifies for each process p_k its status $Status[k] = st_k \in \{c, a\}$ at state s; if $st_k = c$, then none of the processes from the concurrent group at state s has committed and upon operation invocation from p_k at state s, automaton F_{gp} sends to p_k corresponding response which is not an abort event A^k; if $st_k = a$, then some process has committed before p_k and the automaton sends an abort event to p_k when p_k invokes some operation. This means that some processes has committed while being in the same concurrent group as p_k and F_{gp} has not sent an abort event to p_k yet. Therefore if p_k invokes an operation at state s and $st_k = a$, then F_{gp} sends abort event A^k as a response. Initially every st_k has value c, when automaton F_{gp} sends commit event C^k to some process p_k, then the automaton changes the value of $st_{k'}$ to a for every process $p_{k'}$ from the set CP except p_k. When process $p_{k'}$ receives abort event $A^{k'}$, then the automaton changes $st_{k'}$ to c.
- Subset $CP \subseteq P$ is the set of processes such that all processes from CP are concurrent to each other at state s and none of them has committed, i.e. for every process $p_k \in CP$ its status st_k is c. Initially $CP = \emptyset$. When process p_k with $st_k = c$ invokes a read or write operation, process p_k is added to CP (if p_k is already in CP, then CP does not change). When some process $p_k \in CP$ commits, then the automaton empties set CP, i.e. $CP = \emptyset$. Note that there could be other processes which are concurrent to processes from the set CP at state s, but which do not belong to CP since their statuses are a.
- Array Val is a two-dimensional array of current variables, each element $Val[i][j]$ is a variable $v_{i,j}$ that corresponds to process p_i and t-variable x_j, when process p_i reads from t-variable x_j at state s, then the automaton returns to p_i value $v_{i,j}$, when process p_i writes value v to x_j, then the automaton makes a transition to the state with $v_{i,j}$ equal to v. If some process p_i commits, then for every other process p_k and every t-variable x_j the automaton changes $v_{k,j}$ to $v_{i,j}$.
- Function $f : P \to Inv \cup \{\perp\}$ is a pending function which specifies for each process if the process was returned a response after its last invocation. Namely, if $f(p_k) =\perp$ then process p_k was returned a response and p_k can send an invocation event at state s; if $f(p_k) = e$, where $e \in Inv_k$, then process p_k was not returned a response from e and p_k cannot send an invocation event at state s. Initially, $f(p_k) =\perp$ for every process $p_k \in P$.

Formally, we construct I/O automaton $F_{gp} = (St, I, O, s_0, R)$ using the following rules:

- $S = \{s | s = (Status, CP, Val, f)\}$, where:
  - $\forall k \in K, Status[k] \in \{c, a\}$
  - $CP \subseteq P$
  - $\forall k \in K, \forall j \in J, Val[k][j] \in V$, where J is the set of t-variable identifiers
  - $f : P \to Inv \cup \{\perp\}$
- $I = \{x_j.write^k(v) | x_j \in X, k \in K, v \in V\} \cup \{x_j.read^k() | x_j \in X, k \in K\} \cup \{tryC^k | k \in K\}$
- $O = \{v^k | v \in V, k \in K\} \cup \{ok^k | k \in K\} \cup \{C^k | k \in K\} \cup \{A^k | k \in K\}$.
- $s_0 = (Status, CP, Val, f)$ such that $\forall k \in K, Status[k] = c$; $CP = \emptyset$; $\forall k \in K, \forall j \in J, Val[k][j] = 0$; and $\forall p_k \in P, f(p_k) = 0$.

Transition relation $T \subseteq S \times I \times O \times S$ is defined by the following rules:

- $\forall k \in K, \forall x_j \in X, \forall v \in V$, $(s, x_j.write^k(v), s') \in T$ iff all of the following hold:
  - $s = (Status, CP, Val, f)$, $f(p_k) =\perp$
  - $s' = (Status', CP', Val', f')$, $Status' = Status$, $CP' = CP \cup \{p_k\}$, $f'(p_k) = x_j.write^k(v)$ and $f'(p_{k'}) = f(p_{k'})$ for every $p_{k'} \in P \setminus \{p_k\}$
  - Val' is derived from Val by updating the value of $Val[k][j]$ to v

- $\forall k \in K$, $(s, ok^k, s') \in T$ iff all of the following hold:
  - $s = (Status, CP, Val, f)$, $Status[k] = c$ and $f(p_k) = x_j.write^k(v)$ for some $x_j \in X$
  - $s' = (Status', CP', Val', f')$, $Status' = Status$, $CP' = CP$, $f'(p_k) = \perp$ and $f'(p_{k'}) = f(p_{k'})$ for every $p_{k'} \in P \setminus \{p_k\}$
  - $Val' = Val$
- $\forall k \in K, \forall x_j \in X$, $(s, x_j.read^k, s') \in T$ iff all of the following hold:
  - $s = (Status, CP, Val, f)$, $f(p_k) = \perp$
  - $s' = (Status', CP', Val', f')$, $Status' = Status$, $CP' = CP \cup \{p_k\}$, $f'(p_k) = x_j.read^k$ and $f'(p_{k'}) = f(p_{k'})$ for every $p_{k'} \in P \setminus \{p_k\}$
  - $Val' = Val$
- $\forall k \in K, v \in V$, $(s, v^k, s') \in T$ iff all of the following hold:
  - $s = (Status, CP, Val, f)$, $Status[k] = c$ and $f(p_k) = x_j.read^k$ for some $x_j \in X$
  - $s' = (Status', CP', Val', f')$, $Status' = Status$, $CP' = CP$, $f'(p_k) = \perp$ and $f'(p_{k'}) = f(p_{k'})$ for every $p_{k'} \in P \setminus \{p_k\}$
  - $Val' = Val$ and $v = Val[k][j]$
- $\forall k \in K$, $(s, tryC^k, s') \in T$ iff all of the following hold:
  - $s = (Status, CP, Val, f)$, $f(p_k) = \perp$
  - $s' = (Status', CP', Val', f')$, $Status' = Status$, $CP' = CP \cup \{p_k\}$, $f'(p_k) = tryC^k$ and $f'(p_{k'}) = f(p_{k'})$ for every $p_{k'} \in P \setminus \{p_k\}$
  - $Val' = Val$
- $\forall k \in K, v \in V$, $(s, C^k, s') \in T$ iff all of the following hold:
  - $s = (Status, CP, Val, f)$, $Status[k] = c$ and $f(p_k) = tryC^k$
  - $s' = (Status', CP', Val', f')$, $Status'[k] = c$ and $Status'[k'] = a$ for every $k' \in K \setminus \{k\}$, $CP' = \emptyset$, $f'(p_k) = \perp$ and $f'(p_{k'}) = f(p_{k'})$ for every $p_{k'} \in P \setminus \{p_k\}$
  - $Val'[k'][j] = Val[k][j]$ for every $k' \in K$ and every $j \in J$
- $\forall k \in K, v \in V$, $(s, A^k, s') \in T$ iff all of the following hold:
  - $s = (Status, CP, Val, f)$, $Status[k] = a$ and $f(p_k) \neq \perp$
  - $s' = (Status', CP', Val', f')$, $Status'[k] = c$ and $Status'[k'] = Status[k']$ for every $k' \in K \setminus \{k\}$, $CP' = CP$, $f'(p_k) = \perp$ and $f'(p_{k'}) = f(p_{k'})$ for every $p_{k'} \in P \setminus \{p_k\}$
  - $Val' = Val$

For illustration, Figure 15 depicts the states of automaton F_{gp} when $P = \{p_1\}$, $X = \{x\}$, and $V = \{0, 1\}$. Note that the automaton of Figure 15 has no abort events, since process p_1 has no concurrent processes to it. Figure 16 depicts an example history H_{ex} of F_{gp} for three processes $\{p_1, p_2, p_3\}$ and two t-variables $\{x, y\}$.

Theorem 3. *The TM implementation represented by F_{gp} ensures both opacity and global progress in any fault prone system.*

Proof. **Opacity.** Consider any finite history H of F_{gp}. We complete history H by aborting every transaction which is neither committed nor aborted. We denote the resulting history by $com(H)$.

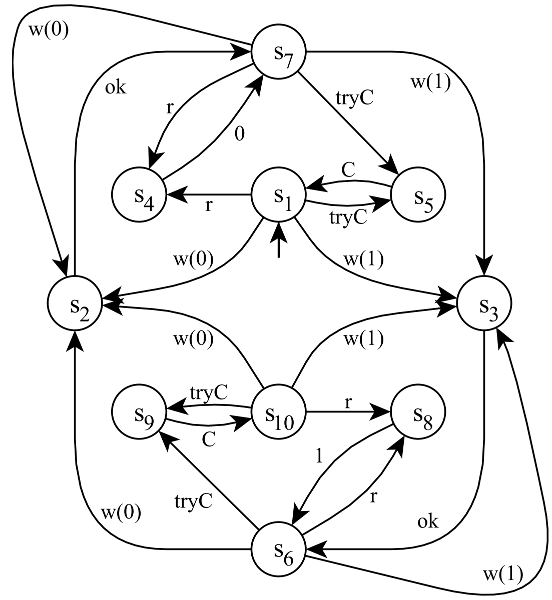

Figure 15. Automaton F_{gp} for a single process p_1 and a single binary t-variable x. For simplicity, r stands for $x.read^1$, $w(1)$ for $x.write^1(1)$, $w(0)$ for $x.write^1(0)$, 1 for 1^1, 0 for 0^1, $tryC$ for $tryC^1$, and C for C^1. The automaton has the following states with s_1 as the initial state:
$s_1 = (c, \emptyset, 0, f(p_1) = \perp)$
$s_2 = (c, \{p_1\}, 0, f(p_1) = x.write^1(0))$
$s_3 = (c, \{p_1\}, 1, f(p_1) = x.write^1(1))$
$s_4 = (c, \{p_1\}, 0, f(p_1) = x.read^1)$
$s_5 = (c, \{p_1\}, 0, f(p_1) = tryC^1)$
$s_6 = (c, \{p_1\}, 1, f(p_1) = \perp)$
$s_7 = (c, \{p_1\}, 0, f(p_1) = \perp)$
$s_8 = (c, \{p_1\}, 1, f(p_1) = x.read^1)$
$s_9 = (c, \{p_1\}, 1, f(p_1) = tryC^1)$
$s_{10} = (c, \emptyset, 1, f(p_1) = \perp)$

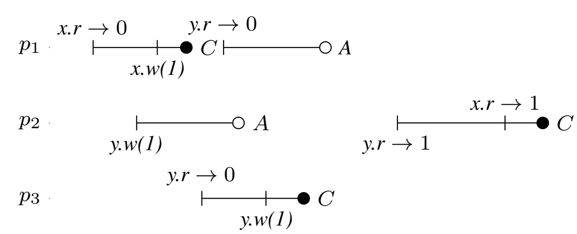

Figure 16. A history H_{ex} of the implementation F_{gp} for three processes and two binary t-variables. For simplicity, $x.r \rightarrow v$ means that a process reads value v from x, $y.r \rightarrow v$ means that a process reads value v from y, $x.w(v)$ means that a process writes value v to x, and $y.w(v)$ means that a process writes value v to y.

Let there be n commit events in history $com(H)$. Denote as C_i the i-th commit event in $com(H)$. Let $s_i = (Status_i, CP_i, Val_i, f_i)$ be a state of F_{gp} at which the i-th commit event was issued. With each C_i we can associate a corresponding set of transactions $Tr_i = \{T_1^i, \ldots, T_{|CP_i|}^i\}$ such that every transaction $T_k^i \in Tr_i$ is a transaction executed by process $p_k \in CP_i$ when automaton F_{gp} is at state s_i.

Consider the following sequential history $H_s = H_{s,1} \cdot \ldots \cdot H_{s,n} \cdot H_{s,n+1}$, such that for each $i \in \{1, \ldots, n\}$, $H_{s,i}$ is formed by concatenating transactions Tr_i in a such way that $H_{s,i}$ ends with the i-th commit event. The suffix $H_{s,n+1}$ is formed by concatenating the rest of the transactions in $com(H)$ which do not belong to any Tr_i. Then, by definition, the sequential history H_s is equivalent to $com(H)$.

We now show that H_s preserves the real-time order of $com(H)$. Consider any two transactions T_1 and T_2 in $com(H)$ such that $T_1 <_{com(H)} T_2$. Since T_1 is committed there exists some Tr_i such that $T_1 \in Tr_i$. Because $T_1 \in Tr_i$, T_1 is a transaction in subsequence $H_{s,i}$ of sequential history H_s. Suppose there exists some Tr'_i such that $T_2 \in Tr'_i$, i.e. T_2 is a transaction in $H_{s,i'}$ of sequential history H_s. Since $T_1 <_{com(H)} T_2$, then transactions T_1 and T_2 are not concurrent and $i < i'$. Thus, $T_1 <_{H_s} T_2$. If there is no Tr_i such that $T_2 \in Tr_i$, then T_2 is a transaction in subsequence $H_{s,n+1}$ of sequential history H_s and therefore $T_1 <_{H_s} T_2$.

To prove that in history H_s every transaction is legal, assume that some transaction T_i is not legal. Since T_i is not legal, then for some t-variable x_j, $visible(T_i)$ does not respect the semantics of x_j. In other words, within some transaction in $visible(T_i)$ some process p_k is returned v^k after a read from x_j, while the value of x_j is not v. Since H_s is equivalent to $com(H)$, then in the history H process p_k receives v^k after invoking a read request on t-variable x_j whose value is v. However, by definition of F_{gp}, F_{gp} can only return the value stored in $Val[k][j]$ which is the value of x_j seen by process p_k: a contradiction.

Global progress. By definition a TM implementation ensures global progress iff in every infinite history H of the corresponding automaton at least one correct process is not pending. Consider any infinite history H of F_{gp} such that some processes are correct in H. Assume that all the correct processes are pending in H. That is there exists some point in time, and correspondingly a prefix H'' of H, after which every correct process will never receive a commit event. Consider any prefix H' of H such that H'' is a prefix of H' and H' takes the automaton to some state $s = (Status, CP, Val, f)$, where for every correct process p_k we have $Status[k]_k = c$. Since after state s none of the correct processes receives a commit event and every p_k has $Status[k]_k = c$, then none of the processes ever invokes a commit request after s or receives an abort event. Then, by definition, all the processes are not correct: a contradiction.

□

7. Concluding Remarks

We propose a framework to formally reason about liveness properties of TMs and introduce the very notion of a TM-liveness property. We prove in particular that in a system with faulty processes (crashes or parasitic), local progress cannot be ensured together with opacity, the safety property typically ensured by most TMs. In other words, we cannot ensure starvation for all and consistency. We presented this impossibility result in its direct and then general form.

Local progress of transactional memory implementations is analogous to wait-freedom in concurrent computing which is the ultimate classical liveness property (for non-transactional objects) in concurrent computing. Just like wait-freedom makes sure processes do not wait for each other, local progress ensures that transactions do not wait for each other. The fact that wait-freedom was shown to be possible to implement led researchers to focus on how to achieve it efficiently. The fact that local progress is impossible to implement means that researchers have to find alternatives. We pointed out a way to circumvent it by weakening a TM-liveness property, requiring only progress of some processes. Other possible ways are weakening a safety property or assuming that a TM implementation has control over the application employing the TM implementation.

As we pointed out, this paper is a first step towards understanding the liveness of TMs and many problems are open. It would be interesting to determine precisely the strongest liveness property that can be ensured by a TM as well as study the impact on the impossibility of reducing the number of possible faults that a TM can face. Another possible direction for future work would be to generalize the impossibility result even further by considering classes of TM-liveness properties that guarantee progress for processes with higher priority.

References

[1] Doherty, S., Groves, L., Luchangco, V., and Moir, M.: Towards formally specifying and verifying transactional memory. REFINE, 2009.

[2] Imbs, D., Mendivil, J.R., and Raynal, M.: Brief announcement: virtual world consistency: a new condition for STM systems. PODC, 2009.

[3] Harris, T., Larus, J. R., and Rajwar, R.: Transactional Memory, 2nd edition. Morgan and Claypool, 2010.

[4] Herlihy, M., and Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. ISCA, 1993.

[5] Shavit, N., and Touitou, D.: Software transactional memory. PODC, 1995.

[6] Abadi, M., Birrell, A., Harris, T., and Isard, M.: Semantics of transactional memory and automatic mutual exclusion. POPL, 2008.

[7] Jagannathan, S., Vitek, J., Welc, A., and Hosking, A.: A transactional object calculus. Science of Computer Programming, 57(2):164186, 2005.

[8] Menon, V., Balensiefer, S., Shpeisman, T., AdlTabatabai, A.R., Hudson, R. L., Saha, B., and Welc, A.: Practical weak-atomicity semantics for Java STM. SPAA, 2008.

[9] Moore, K. F., and Grossman, D.: High-level small-step operational semantics for transactions. POPL, 2008.

[10] Alpern, B., and Schneider, F.B.: Defining liveness. Inf. Process. Lett., 21(4):181185, 1985.

[11] Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems, 13(1):124149, 1991.

[12] Fetzer, C.: Robust transactional memory and the multicore system model. DISC09 workshop WTTM, 2009.

[13] Fraser, K.: Practical Lock-Freedom. PhD thesis, University of Cambridge, 2003.

[14] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W.N.: Software transactional memory for dynamic-sized data structures. PODC, 2003.

[15] Dice, D., Shalev, O., and Shavit, N.: Transactional locking II. DISC, 2006.

[16] Dragojević, A., Guerraoui, R., and Kapałka, M.: Stretching transactional memory. PLDI, 2009.

[17] Felber, P., Riegel, T. and Fetzer, C.: Dynamic performance tuning of word-based software transactional memory. PPoPP, 2008.

[18] Guerraoui, R., and Kapałka, M.: Principles of Transactional Memory. Morgan and Claypool, 2010.

[19] Papadimitriou, C. H.: The serializability of concurrent database updates. Journal of the ACM, 26(4), pp. 631–653, 1979.

[20] Herlihy, M., Shavit, N.: On the nature of progress. DISC11 workshop WTTM, 2011.