Liveness in Transactional Memory

Victor Bushkov and Rachid Guerraoui

Abstract In this chapter we give a formal overview of liveness properties of transactional memory (TM) systems. Unlike safety properties, which require some ’bad’ events not to occur, liveness properties require some ’good’ events to eventually occur. Usually, liveness properties of shared memory systems require some operations to eventually return a response (terminate). However, in the context of TM systems operation termination is not enough to ensure meaningful progress. It is necessary to require some transactions to eventually commit. In this chapter we give precise definitions of liveness properties and what it means for a TM systems to satisfy a liveness property. Using the defined formal framework we give some impossibility results. We show that it is impossible to guarantee both local progress, the strongest TM liveness property that requires every correct transaction to eventually commit, and common TM safety properties such as strict serializability or opacity in a fault prone system.

1 Introduction

Transactional memory (TM) [13, 16, 26] is a concurrency control paradigm that aims at simplifying concurrent programming. It provides non-expert programmers with an abstraction, called *transaction*, such that transactions concurrently execute atomic pieces of sequential code of some application. Each transaction is executed by some process (thread) and contains transactional operations. A transactional operation is either an access (read or write) to a transactional variable (data item) or a request to commit the transaction. If the transaction is committed, then the effects

Victor Bushkov
EPFL, IC, LPD e-mail: `victor.bushkov@epfl.ch`

Rachid Guerraoui
EPFL, IC, LPD e-mail: `rachid.guerraoui@epfl.ch`

of its operations become visible to subsequent transactions, and if it is aborted, then the effects are rolled back. 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.

Most of the work on the theory of transactional memory focused solely on *safety* (consistency), i.e., on what TMs *should not do*. Indeed, correctness conditions for TMs have been proposed in [11, 18, 5, 6, 8] and programming language level semantics of specific classes of TM implementations have been determined, e.g., in [1, 19, 22, 23]. Most 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 aborting all operations. To be meaningful, a TM has to ensure that some transactions should eventually commit which is captured by a *liveness* property [2].

Generally, in shared-memory systems, a liveness property states when a certain process that invokes an operation on a shared object is guaranteed to return from this operation, i.e. makes progress [17]. One of the widely studied such property is *wait-freedom* [14]. It ensures, intuitively, that *every* process invoking an operation on a shared object 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 and forms the consensus number hierarchy of shared objects [14]. However, requiring TM systems to ensure only wait-freedom would, however, not be enough to ensure any meaningful progress: processes of which all transactions are *aborted* might be satisfying wait-freedom (since every transactional operation returns a response) but would not be making any real progress. To ensure meaningful progress, a TM liveness property should require transaction *commitment*, beyond operation *termination*. In other words, it should require certain processes to eventually commit some of their transactions. One would expect from a TM that every process that keeps executing transactions eventually commits some of them—a property that we call *local progress* and that is similar in spirit to wait-freedom. Not satisfying this property means that some processes might never commit any of their transactions starting from some point in time.

A TM implementation that protects every transaction using a single fair global lock could ensure local progress: such a TM would execute all transactions sequentially, thus avoiding conflicts between transactions. 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, 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. Hence, to be really meaningful a TM liveness property should enforce some ”independent” progress.

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 and can fail by

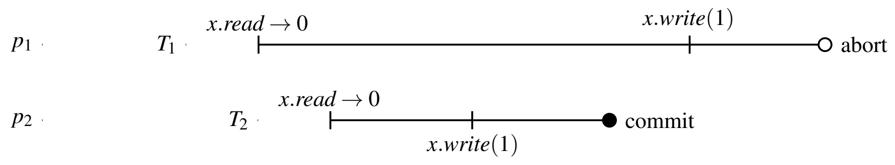

Fig. 1 An illustration of the difficulty of ensuring local progress. The scenario can be repeated infinitely many times preventing transaction T_1 from ever committing.

crashing. A TM implementation that is resilient to crashes enables the progress of a process even if other processes are suspended for a long time or crashed.

However, resiliency against crashes is not enough. Consider a transaction that holds a global lock which does not crash and never invokes a commit request. Such a transaction would prevent all other transactions from making progress. Therefore, 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, shown 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, writes value 1 to x, and attempts to 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 T_2. But then, if process p_1 writes value 1 to x and attempts to commit 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 T_2, forcing process p_1 to abort T_1 later.

We consider a set-based definition of liveness, i.e. we consider a *TM-liveness* property L as a set of fair histories, so that a TM implementation ensures the property if every fair history of the implementation belongs to L. A history is basically a sequence of invocations and responses of operations executed within transactions, and a fair history is a history augmented with *crash* events. The focus on fair histories is necessary because a TM-liveness property should not require progress from processes which do not take any steps in an execution, i.e. crash in that execution. So, to distinguish crashed processes from processes that take infinitely many steps without returning a response of a transactional operation, we augment histories with crash events. Like fairness properties are defined in [27], we define a TM-liveness property as a weakening of local progress, which has the strongest progress requirement among TM-liveness properties.

Since safety properties state that some events should not occur and liveness properties state that some events should eventually occur, safety and liveness requirements might contradict each other. A safety requirement may make it impossible to guarantee a liveness requirement and vice versa. The question is, under what conditions which safety and liveness properties are impossible to guarantee? We address this question in the TM context by proving an impossibility result which states that no TM implementation can ensure both *local progress* and *opacity* in any 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 (even aborted or live) 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, eventually commits its transactions. In fact, we prove a more general result stating that no TM implementation can ensure any safety property that is at least as strong as strict serializability together with the progress of at least two correct processes and any correct process that runs alone.

2 Preliminaries

2.1 System model

We consider a system of *n asynchronous processes* $p_1, \dots, p_n$ that communicate with each other by executing operations on *shared 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 a set of *base objects*. Base objects are shared objects which are accessed via *atomic* operations called *primitives*.

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. When a process p_i invokes an operation op on a shared object O, then p_i follows the implementation of O, possibly accessing some number of base objects and executing local computations, until p_i is returned the 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 on shared objects operations are called (invocation and response) *events*.

2.2 Histories and executions

Let I be an implementation of a shared object O. A *configuration* C of I determines the current state of each process and of each base object used in I. The *initial configuration* C_0 of I is a configuration when all processes and all base objects are at their initial states. A *step* s (executed by some process p_i) of I can be one of the following: (i) an invocation event of some operation on O, (ii) a response event of some operation from O, (iii) a single primitive operation and one or more computations local to p_i. An *execution* $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \ldots$ of I is a (finite or infinite) sequence of alternating configurations and steps of I such that: (i) C_0 is the initial configuration, and for any C_i, s_i, and C_{i+1} in α the execution of step s_i by I at configuration C_i results in the new configuration C_{i+1}. We define a *projection* $\alpha|p_k$ of an execution α on a process p_k as the longest subsequence of α consisting only of steps of p_k.

The order in which processes take steps is determined by a *scheduler*. Processes and TM implementations have no control over a scheduler. The scheduler decides which process is allowed to execute a step at a given point in time. These decisions form a *schedule* which is a finite or an infinite sequence of process identifiers.

The longest subsequence of an execution α of I consisting only of invocation and response events is called a *history* of I, and is denoted by H_α. We define a *projection* $H|p_k$ of a history H on a process p_k as the longest subsequence of H consisting of invocation and response events associated with p_k.

2.3 Transactional Memory

Transactional memory allows processes to execute pieces of sequential code within transactions. The code contains accesses to *transactional variables* (t-variables for short) which represent shared data. For presentation simplicity, we focus on t-variables that support *read* and *write* operations. Let K be the set of *process identifiers*, $P = \{p_k | k \in K\}$ be the set of processes, and let X be the set of *t-variables*. Each t-variable can take values from a set V. To write a value v to a t-variable x process p_k invokes $x.write^k(v)$ and receives as a response either ok, if the write was successful, or an abort event A^k if the transaction has to be aborted. To read a value from a t-variable x process p_k invokes $x.read^k$ and receives as a response either the value of t-variable v or an abort event A^k if the transaction has to be aborted. To commit a transaction process p_k invokes a commit request $tryC^k$ and receives as a response either a commit event C^k or an abort event A^k. 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. Also, let $Inv = \cup_{k \in K} Inv_k$ and $Res = \cup_{k \in K} Res_k$. Usually TM implementations provide additional transactional operations such as the request to start a transaction, the request to create a new t-variable (in the case of dynamic TMs), and a request to abort a transaction. Our theoretical results hold for TM implementations that provide these operations. However, for simplicity, we assume TM implementations that provide only operations to read/write a t-variable and commit a transaction.

Denote by Σ_k a set such that $\Sigma_k = \{x.write^k(v) \cdot ok^k | x.write^k(v) \in Inv_k\} \cup \{x.read^k() \cdot v^k | x.read^k() \in Inv_k \text{ and } v^k \in Res_k\} \cup \{tryC^k \cdot C^k\} \cup \{inv \cdot A^k | inv \in Inv_k\}$, i.e. Σ_k contains concatenations of invocations and their possible responses associated with process p_k. Also, let Σ_k^∞ be the set of all finite and infinite sequences over Σ_k. A history H of a TM implementation is *well-formed* if for every $p_k \in P$ either $H|p_k \in \Sigma_k^\infty$ or $H|p_k \in \Sigma_k^* \cdot Inv_k$ holds, i.e. $H|p_k$ is a sequence of alternating invocation and response events. In the rest of the chapter we assume only well-formed histories.

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

- either e_1 is the first event in $H|p_k$, or the event e' which precedes e_1 in $H|p_k$ is either A^k or C^k, and
- e_m is either A^k or C^k or the last event in $H|p_k$, and
- no event in T, except e_m, 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 precedes the first event of T_2 in H. Transactions T_1 and T_2 are *concurrent* if T_1 does not precede T_2 and T_2 does not precede T_1. 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 by invoking concurrently requests (read, write, and commit requests) and receiving corresponding responses from the implementation. Processes send commit requests 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 force the TM implementation to abort or delay some 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.

2.4 Process Failures

Let α be an infinite execution. Process p_k *crashes* in α if $\alpha|p_k$ is finite. That is, a process crashes in an infinite execution if it stops taking steps in the execution.

Intuitively, a parasitic process is a process that keeps executing transactional operations but, from some point in time, never attempts to commit (by invoking operation $tryC$) when given a chance to do so. Note that if starting from some moment in time every transaction executed by the process is prematurely aborted, i.e. aborted before the process invokes a commit request, in general, we cannot tell whether the process intended to eventually invoke a commit request or not. Therefore, we consider such processes as not parasitic.

Let α be an infinite execution. Process p_k is *parasitic* in α, if there is a suffix α' of α such that: (i) p_k executes infinitely many transactional operations in α', (ii) α' does not contain A^k events, and (iii) α' does not contain $tryC^k$ requests.

Process p_k is *correct* in an infinite execution α if p_k is not parasitic in α and does not crash in α.

We define a *crash-prone system* (respectively, *parasitic-prone system*) Sys to be a system of processes in which any process can crash (respectively, be parasitic). A *fault-prone system* Sys is a system which is crash-prone or parasitic-prone. Note that a fault-prone system can have both crashed and parasitic processes.

2.5 Safety properties of TM

Intuitively a safety property of TM implementations should capture the fact that 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. We consider two safety properties of TM implementations: strict serializability and opacity. Intuitively, strict serializability requires every committed transaction to observe a consistent state of the system [24], while opacity requires every transaction (even aborted or unfinished) to observe a consistent state of the system [12].

We say that history H is *equivalent* to history H' if for every process $p_k \in P$ we have $H|p_k = H'|p_k$. A transaction T in history H is *commit-pending* if T ends with a commit request *tryC*. A transaction T in history H is *live* if T is not commit-pending, aborted, or committed. We obtain a completion of a finite history H by aborting every live transaction and by committing or aborting every commit-pending transaction. Formally a *completion* $comp(H)$ of a history H is a history derived from H by appending the following events:

- for every live transaction T (executed by p^k) we append $tryC^k \cdot A^k$
- for every commit pending transaction T (executed by p^k) we append either C^k or A^k.

If $comp(H) = H$, then H is a *complete* history. We say that a history H' *preserves the real time order* of a 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_i in the subsequence, either $j = i$ or $T_i <_{H_s} T_j$. Transaction T_j is *legal* in H_s if for every t-variable $x \in X$ history $visible(T_j)$ respects the sequential specification of x, i.e. for every transaction T_i in $visible(T_j)$ and every response event v^k in T_i, v is the value of the previous write to x invocation event within a committing transaction in $visible(T_j)$ or v is the initial value of x if there are no write to x invocation events within any committing transaction in $visible(T_j)$ before v^k.

A finite history H is *opaque*[1] if there exists a sequential history H_s equivalent to $comp(H)$, such that H_s preserves the real-time order of $comp(H)$, and every transac-

[1] Since the way we define opacity is not prefix-closed it is not exactly a safety property. However, for the sake of simplicity, we do not consider a prefix-closed definition of opacity since in terms of

tion in H_s is legal. A finite history H is *strictly serializable* if there exists a sequential history H_s equivalent to H', where H' is obtained from H by removing every aborted and live transaction and some of the commit-pending transactions and appending to H a commit event for every commit-pending transaction which is not removed, such that H_s preserves the real-time order of H, and every transaction in H_s is legal. A TM implementation I ensures opacity (respectively, strict serializability) if for every execution α of I, H_α is opaque (respectively, strictly serializable).

For example, the history in Figure 1 is opaque, while the history in Figure 2 is not opaque but strictly serializable.

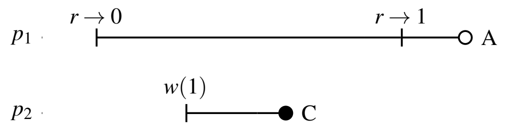

Fig. 2 A history which is not opaque but strictly serializable. All operations access the same t-variable. For simplicity, $r \to v$ denotes both the invocation of a read operation and its response v, $w(v)$ denotes both the invocation of a write operation (with value v) and its response *ok*, C denotes both the invocation of a commit request and a commit event, A denotes both the invocation of a commit request and an abort event

3 Liveness of a TM

3.1 TM-liveness Properties

Basically, a TM-liveness property states whether some process p_k should make progress in some execution α. Clearly, progress cannot be required for crashed or parasitic processes: these processes have executions with a finite number of *tryC* operation invocations. Thus, we should require progress only for correct processes (which basically captures the fairness requirement). Like a fairness property is defined in [27], we define a TM-liveness property as a weakening of the strongest TM-liveness property. The strongest TM-liveness property that we can require of a TM system is to ensure that every correct process makes progress.

Next we introduce the notion of a fair history in order to distinguish a process that crashes from a process that takes infinitely many steps without returning a response when defining a liveness property. We derive a fair history F_α by augmenting a history H_α, of some execution α, with *crash* events. Formally, we derive a *fair history* F_α in the following way: for every process p_k that crashes in α we insert a crash event $crash^k$ between the last event e of p_k and the event that follows after e

TM implementations a prefix-closed definition is equivalent to a non-prefix-closed one (i.e. every TM implementation which ensures non-prefix-closed also ensure a prefix-closed one).

in H_α. A process p_k is *correct* in a fair history F_α, if p_k is correct in α. Herein, if α is clear from the context, we omit α from the notation of a (fair) history H_α and use just H instead. A process p_k *makes progress* in a fair history F, if F contains infinitely many commit events C^k.

A fair history F ensures *local progress* if every correct process makes progress in F, or F does not have any correct processes. Let L_{local} denote the set of all possible fair histories that satisfy local progress. Then, a *TM-liveness* property L is a set of fair histories such that $L_{local} \subseteq L$. Given two TM-liveness properties L_1 and L_2, we say that L_1 is weaker (stronger) than L_2 if $L_2 \subseteq L_1$ ($L_1 \subseteq L_2$). A fair history F ensures a TM-liveness property L iff $H \in L$. A TM implementation I *ensures a TM-liveness property* L if for every execution α of I its corresponding fair history F_α ensures L.

3.2 Examples of TM-liveness Properties

3.2.1 Local Progress

Roughly speaking, a TM implementation I ensures local progress if I guarantees that *every* correct process in a fair history makes progress, i.e. has infinitely many of its transactions committed. Note that local progress requirements also imply the requirement of wait-freedom of individual transactional operations. Therefore, every TM-implementation that ensures local progress also ensures wait-freedom [14], which requires each individual transactional operation to receive a response. However, a TM-implementation might ensure wait-freedom without ensuring local progress, e.g. when all transactional operations receive a response each transaction is aborted.

For example, Figure 3 shows an infinite history which ensures local progress in a system with two processes and one t-variable. Both processes make progress in the history.

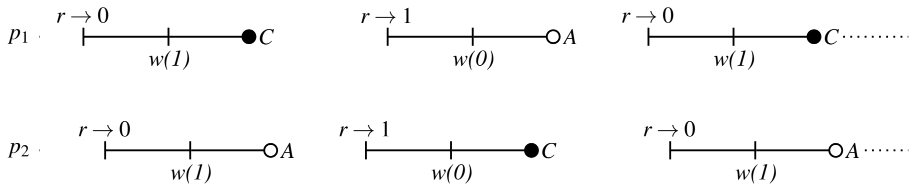

Fig. 3 An infinite fair history with two processes and one t-variable that ensures local progress. Each process executes an infinite number of transactions that either read value 0 and write value 1 or read value 1 and write value 0.

3.2.2 Global Progress

A TM implementation I ensures *global progress* if I guarantees that *some* correct process in a fair history makes progress, i.e. has infinitely many of its transactions committed. Formally, we define global progress, as a TM-liveness property L_{global} such that a fair history F belongs to L_{global} iff at least one correct process in F makes progress in F, or F does not have correct processes. Note that every TM-implementation that ensures global progress also ensures lock-freedom [14], which requires some individual transactional operation to receive a response.

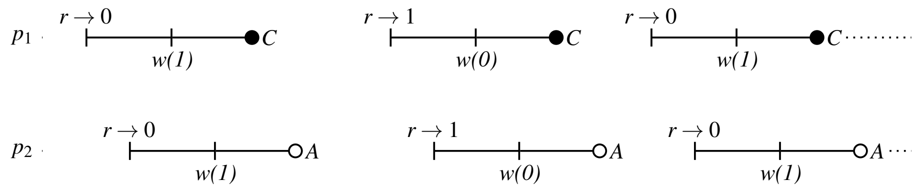

Fig. 4 An infinite fair history with two processes and one t-variable that ensures global progress. Processes execute an infinite number of transactions that either read value 0 and write value 1 or read value 1 and write value 0.

Figure 4 shows an infinite fair history which ensures global progress in a system of 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 implementation I ensures *solo progress* if I guarantees that every correct process which runs alone in a fair history makes progress, i.e. has infinitely many of its transactions committed. A correct process runs alone if starting from some point in time it is the only process that takes steps in an execution. Formally, a process p_k *runs alone* in an infinite fair history F if p_k is correct in F and all other processes crash in F (i.e. stop taking steps in the corresponding execution). We define solo progress, as a TM-liveness property L_{solo} such that a fair history F belongs to L_{solo} iff a process that runs alone in F makes progress in H, or F does not have a process that runs alone in F. Note that every TM-implementation that ensures solo progress also ensures obstruction-freedom [15], which requires each individual transactional operation to receive a response if the operation runs alone.

Figure 5 depicts an infinite fair history which ensures solo progress in a system with three processes and one t-variable. Processes p_1 and p_2 crash, and process p_3 runs alone and makes progress.

Note that according to the definition of solo progress, a transaction which does not encounter step contention with other transactions, i.e. the transaction runs alone, is allowed to abort. This is because solo progress is a liveness property, and there-

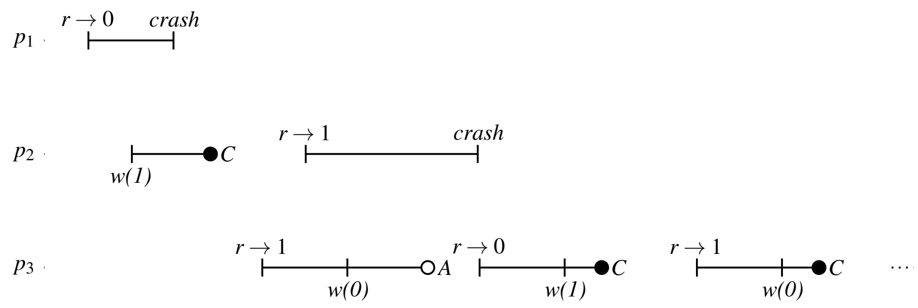

Fig. 5 An infinite fair history with three processes and one t-variable that ensures solo progress. Process p_1 starts a transaction by invoking a read operations, but then it crashes. Process p_2 executes two transactions, but it crashes during the execution of the second transaction. Process p_3 executes an infinite number of transactions that either read value 0 and write value 1 or read value 1 and write value 0.

fore it should allow any possible finite fair history (by the definition of a liveness property [2, 25]). If we change the definition of solo progress so that the new definition requires every transaction which runs alone to commit, then the resulting new definition would not be a liveness property.

Obstruction-free TM implementations [12, 15] ensure solo progress in systems that are not parasitic-prone. Lock-based TM implementations, such as TinySTM [9] and SwissTM [7], ensure solo progress in systems that are not crash-prone. However, lock-based TMs that use lazy acquire, such as TL2 [4], ensure solo progress in systems that are not crash-prone.

Using the same formal framework we can define other kinds of TM-liveness properties. For example, in [3] we define a stronger version of solo progress which requires progress from a process if all other processes either crash or become parasitic starting from some point in time. Basically, such TM-liveness property states that if no other processes attempt to commit their transactions then the only correct process should make progress.

4 Impossibility of Local Progress

Like in any distributed problem, each execution 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 schedule of steps 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 against the implementation. The environment wins the game against a TM implementation, if the resulting infinite fair history violates local progress. To prove the impossibility result, we show a wining strategy for the environment.

Theorem 1. *For every fault-prone system, there does not exist a TM implementation that ensures both local progress and opacity in that system.*

Proof. Assume otherwise, i.e. that there exists a fault-prone system Sys for which there exists a TM implementation I that ensures local progress and opacity in Sys. To find a contradiction, we exhibit a winning strategy (Strategies 1 and 2 below) for the environment resulting in an infinite fair history of I which does not ensure local progress.

By its definition, a fault-prone system Sys is a system in which any number of processes can crash or be parasitic. We thus consider two different cases:

***Sys* is crash-prone.**

Consider two processes p_1 and p_2 and the environment that interacts with I using Strategy 1.

Strategy 1.

1. **Step 1.** Process p_1 invokes a read operation on t-variable x. Only process p_1 takes steps until it receives a response. When p_1 receives a response, which is either v'^1 or A^1, the strategy goes to Step 2.
2. **Step 2.** Process p_2 invokes a read operation on t-variable x and takes steps until it receives as a response v''^2 or A^2. If the response is A^2, then the strategy repeats Step 2. Otherwise p_2 invokes an operation on x, which writes to x either (I) value $v'+1$, if p_1 received v'^1 in Step1, or (II) value $v''+1$, if p_1 received A^1 in Step1, and takes steps until it receives as a response ok^2 or A^2. If the response is A^2, then the strategy repeats Step 2. Otherwise p_2 invokes $tryC^2$ operation and takes steps until it receives a response C^2 or A^2. If the response is A^2, the strategy repeats Step 2. Otherwise the strategy goes to Step 3. Only process p_2 takes steps until it receives C^2 as a response.
3. **Step 3.** If p_1 received A^1 in Step 1, then the strategy goes to Step 1. Otherwise process p_1 resumes taking steps by invoking a write operation on t-variable x which writes value $v''+1$ to x, and then executes until it receives a response. If the response is A^1, then the strategy goes to Step 1. Otherwise p_1 invokes $tryC^1$ operation and executes the operation until it receives a response. If the response is A^1, the strategy goes to Step 1. Otherwise the strategy stops.

First, we prove that processes p_1 and p_2 cannot be parasitic in any execution corresponding to Strategy 1. This is because Strategy 1 does not have loops in which some process invokes infinitely many operations within the same transaction without ever invoking a commit request or receiving an abort event. Note that according to the strategy, process p_1 can crash when transactions of process p_2 are repeatedly aborted in Step 2. Therefore, the strategy does not describe the behavior of processes in a crash-free system, i.e. system in which no process is allowed to crash.

Next, we show that there exists an infinite fair history F of I corresponding to some execution of I according to Strategy 1. To do so, we prove that Strategy 1 never terminates. We first prove that the individual transactional operations of I are obstruction-free, i.e. we prove that each operation in Strategy 1 eventually returns a response. If in Strategy 1 some process p_k, where $k \in \{1,2\}$, executing a transactional operation, does not return a response, then p_k takes infinitely many steps, and consequently p_k is correct. However, p_k does not make progress: a contradiction to the fact that I ensures local progress. Since individual operations of the implementation are obstruction-free, then the strategy terminates iff at Step 3 process p_1 is returned C^1 by I.

Assume some finite history H_f of I corresponding to an execution according to Strategy 1 such that the last event in H_f is C^1 (Figure 6). Since I ensures opacity, there exists a sequential finite history H_s which is equivalent to $comp(H_f)$, preserves the real-time order of $comp(H_f)$, and every transaction in H_s is legal. Since history H_f has no transactions which are either live or commit-pending, then $comp(H_f) = H_f$. 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 $v''+1$. 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 $v'+1$. Thus, H_f is not opaque. Since every history H_f of I that ends with a commit event C^1 is not opaque and I ensures opacity, then H_f is not a history of I corresponding to Strategy 1. In other words, every history of I corresponding to some execution according to Strategy 1 is infinite.

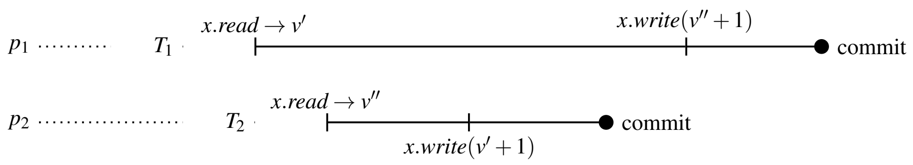

Fig. 6 A suffix of history H_f corresponding to an execution according to Strategy 1 (and Strategy 2) with the last two transactions of p_1 and p_2.

Consider some infinite execution α of I corresponding to Strategy 1. Since process p_1 never receives commit event C^1 from I, then p_1 does not make progress in the corresponding infinite fair history F_α. Since *Sys* is crash-prone, then process p_1 either crashes in α or does not. Therefore, we focus on the following two cases:

- **Process p_1 crashes in α.** According to the strategy, process p_1 crashes in F_α iff starting from some point in time the strategy executes infinitely many iterations of Step 2 without going to Step 3. Since no process can be parasitic in any execution corresponding to Strategy 1 and p_2 takes infinitely many steps in α, process p_2 is correct in F_α. Since I ensures local progress and p_2 is correct in F_α, then process p_2 eventually receives commit event C^2 in Step 2, and therefore the strategy should eventually go to Step 3: a contradiction.

- **Process p_1 does not crash in α.** Since p_1 cannot be parasitic in α, then p_1 is correct in F_α. Since I ensures local progress, then p_1 makes progress in F_α: a contradiction.

***Sys* is parasitic-prone.** Consider two processes p_1 and p_2 and the environment that interacts with I using the following strategy:
Strategy 2.

1. **Step 1.** Process p_1 invokes a read operation on t-variable x and takes steps until it receives as a response v'^1 or A^1. Then process p_2 invokes a read operation on x and takes steps until it receives as a response v''^2 or A^2. If the response is A^2, then the strategy repeats Step 1. Otherwise p_2 invokes a write operation which writes to x either (I) value $v'+1$, if p_1 received v'^1, or (II) value $v''+1$, if p_1 received A^1, and then p_2 takes steps until it receives a response. If the response is A^2, then the strategy repeats Step 1. Otherwise p_2 invokes $tryC^2$ operation and takes steps until it receives a response. If the response is A^2, then the strategy repeats Step 1. Otherwise the strategy goes to Step 2.
2. **Step 2.** If p_1 received A^1 in Step 1, then the strategy goes to Step 1. Otherwise process p_1 invokes a write operation on x which writes value $v''+1$ to x, and p_1 takes steps until it receives a response. If the response is A^1, then the strategy goes to Step 1. Otherwise p_1 invokes $tryC^1$ operation and takes steps until it receives a response. If the response is A^1, then the strategy goes to Step 1. Otherwise the strategy stops.

We first prove that the individual transactional operations of I are obstruction-free, i.e. we prove that each operation in Strategy 2 eventually returns a response. If in Strategy 1 some process p_k, where $k \in \{1,2\}$, executing a transactional operation, does not return a response, then p_k takes infinitely many steps, and consequently p_k is correct. However, p_k does not make progress: a contradiction to the fact that I ensures local progress. Because the individual transactional operations are obstruction-free and because both processes take steps before Step 1 in Strategy 2 can be repeated, processes p_1 and p_2 cannot crash in any execution corresponding to Strategy 2. Note that according to the strategy, process p_1 can become parasitic when transactions of process p_2 are repeatedly aborted in Step 1 and the read operation of p_1 is never aborted. Therefore, the strategy does not describe the behavior of processes in a parasitic-free system, i.e. system in which no process is allowed to be parasitic.

Next, we prove that Strategy 2 never terminates, i.e. that at Step 2 process p_1 is never returned C^1 by I in any history of I corresponding to an execution of the strategy. Assume some finite history H_f of I corresponding to an execution of Strategy 2 such that the last event in H_f is C^1 (Figure 6). Since I ensures opacity, there exists a sequential finite history H_s which is equivalent to $comp(H_f)$, preserves the real-time order of $comp(H_f)$, and every transaction in H_s is legal. Since history H_f has no transaction which are either live or commit-pending, then $comp(H_f) = H_f$. 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 $v''+1$. 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 $v'+1$. Thus, H_f is not opaque. Since every history H_f of I that ends with commit event C^1 is not opaque and I ensures opacity, then H_f is not a history of I corresponding to the execution of the strategy. In other words, every history of I corresponding to the execution of Strategy 2 is infinite.

Consider now some infinite execution α of I corresponding to the execution of the above strategy. Since process p_1 never receives commit event C^1 from I, then p_1 does not make progress in the corresponding infinite fair history F_α. Since *Sys* is parasitic-prone, then process p_1 is either parasitic in α or not. Therefore, we focus on the following two cases:

- **Process p_1 is parasitic in α.** According to the strategy, process p_1 is parasitic in F_α iff starting from some point in time the strategy executes infinitely many iterations of Step 1 without going to Step 2. Strategy 2 repeats Step 1 without going to Step 2 iff process p_2 is repeatedly returned abort event A^2 in Step 1. Since no process can crash in any execution corresponding to Strategy 1 and p_2 receives infinitely many abort events in α, process p_2 is correct in F_α. Since I ensures local progress and p_2 is correct in F_α, then process p_2 shoudl eventually receive commit event C^2 in Step 1, and therefore the strategy should eventually go to Step 2: a contradiction.
- **Process p_1 is not parasitic in α.** Since p_1 does not crash in α, p_1 is correct in F_α. Since I ensures local progress, then p_1 makes progress in F_α: a contradiction.

□

5 Generalizing the Impossibility

In this section we generalize the impossibility result of the previous section. Namely, we determine a larger class of TM-liveness properties that are impossible to implement together with strict serializability, which is weaker than opacity, in fault-prone systems.

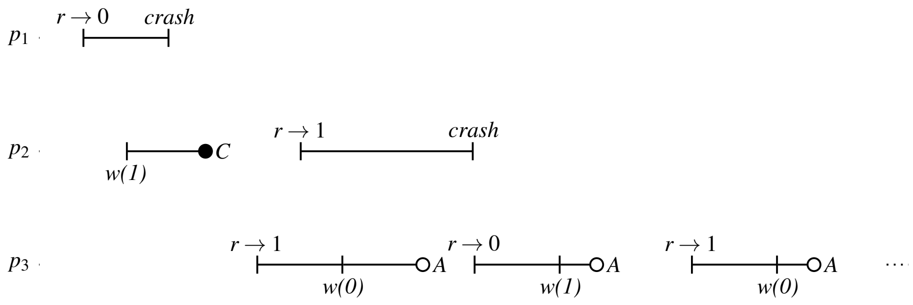

Fig. 7 An infinite fair history with three processes and one t-variable that does not ensure any non-blocking TM-liveness property. Process p_1 starts a transaction by invoking a read operations, but then it crashes. Process p_2 executes two transactions, but it crashes during the execution of the second transaction. Process p_3 executes an infinite number of transactions which read value 0 (read value 1) and write value 1 (write value 0).

5.1 Classes of TM-liveness properties

Non-blocking TM-liveness properties.

Intuitively, we say that a TM-liveness property is non-blocking if it guarantees progress for every correct process that eventually runs alone. More precisely, a TM-liveness property L is *non-blocking* iff L is stronger than L_{solo}.

For example, Figure 3, Figure 4, and Figure 5 show infinite fair histories which ensure non-blocking TM-liveness properties while Figure 7 shows an infinite fair history which does not ensure any non-blocking TM-liveness property. Local progress, global progress, and solo progress are non-blocking. Note that solo progress is the weakest among non-blocking TM-liveness properties while local progress is the strongest among non-blocking properties.

Biprogressing TM-liveness properties.

Intuitively, we say that a TM-liveness property L is a biprogressing property if it requires that at least two correct processes make progress. More precisely, a TM-liveness property L is *biprogressing* if for every $F \in L$ at least two processes are correct in F, only if at least two processes make progress in F.

For example, Figure 3 and Figure 5 show infinite fair histories which ensure a biprogressing property while Figure 4 shows an infinite fair history which does not ensure any biprogressing property. Local progress is a biprogressing property while global progress and solo progress are not biprogressing.

5.2 Generalized Result

In this section we show that TM-liveness properties that are both non-blocking and biprogressing are impossible to implement together with strict serializability in any fault-prone system. We start by stating the following lemma, which says, intuitively, that there exists a fair history in which a process executing infinitely many transactions can block the progress of all other processes if the TM ensures any non-blocking TM-liveness property. The proof of the lemma follows the same line of reasoning as the proof of Theorem 1.

Lemma 1. *For any fault-prone system and every TM implementation that ensures strict serializability and a non-blocking TM-liveness property in that system, there exists an infinite fair history F of the implementation such that at least two processes are correct in F and at most one process makes progress in F.*

Proof. Let I be a TM implementation ensuring strict serializability and a non-blocking TM-liveness property in a fault-prone system Sys. To exhibit a fair 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 of the lemma.

By definition, fault-prone system Sys is a system in which any process can crash or be parasitic. We thus consider two different cases:

Sys is crash-prone. Consider two processes p_1 and p_2 that interact with I. The environment uses Strategy 1 to win the game. We can show that processes p_1 and p_2 cannot be parasitic in any execution corresponding to Strategy 1 and that Strategy 1 never terminates using the arguments as in Theorem 1 (because those arguments do not involve live or aborted transactions).

Consider some infinite execution α of I corresponding to Strategy 1. Since process p_1 never receives commit event C^1 from I, then p_1 does not make progress in the corresponding infinite fair history F_α. Since Sys is crash-prone, then process p_1 either crashes in α or does not.

Assume that process p_1 crashes in fair history F_α. According to the strategy, process p_1 crashes in F_α only if process p_2 invokes infinitely many operations and does not make progress, 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 and p_2 cannot crash, p_2 is correct in F_α. Because p_2 runs alone in F_α and I ensures a TM-liveness property which is non-blocking, then p_2 makes progress in H: a contradiction. Thus, p_1 does not crash in F_α. Since p_1 is not parasitic in α, p_1 is correct in F_α.

According to the strategy, p_2 does not crash in F_α since Step 2 is repeated infinitely often. Since Step 2 and Step 1 are repeated infinitely often (because p_1 does not crash in F_α), then p_2 receives infinitely many commit events C^2, i.e. p_2 is correct. Thus, in fair history F_α both of the processes are correct and at most one process makes progress (since p_1 is never returned C^1).

***Sys* is parasitic-prone.** Consider two processes p_1 and p_2 that interact with I. The environment uses Strategy 2 to win the game. We can show that processes p_1 and p_2 do not crash in any execution corresponding to Strategy 2 and that Strategy 2 never terminates using the same line of reasoning as in Theorem 1.

Consider now some infinite execution α of I corresponding to the execution of the above strategy. Since process p_1 never receives commit event C^1 from I, then p_1 does not make progress in the corresponding infinite fair history F_α. Since *Sys* is parasitic-prone, then process p_1 is either parasitic in α or not.

Assume that p_1 is parasitic in α. According to the strategy, p_1 can be parasitic only if p_2 does not make progress in F_α and is returned A^2 infinitely often (i.e. p_2 is correct in F_α). Since process p_2 runs alone in F_α and I ensures a non-blocking TM-liveness property, then p_2 makes progress in H: a contradiction. Thus, p_1 cannot be parasitic in α. Since p_1 does not crash in α, p_1 is correct in F_α.

Process p_2 cannot be parasitic in α since p_2 either invokes $tryC^2$ or is returned A^2 infinitely often at Step 1. Thus, in history F_α both of the processes are correct and at most one process makes progress (since p_1 is never returned C^1). □

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 non-blocking, then we can find an infinite fair history of any TM implementation in any fault-prone system in which at least two processes are correct and at most one process makes progress: a contradiction. Thus, we have the following theorem.

Theorem 2. *For every fault-prone system and every TM-liveness property L which is non-blocking and biprogressing there is no TM implementation that ensures strict serializability and L in that system.*

6 Conclusion

In this chapter we introduced a set-based framework to formally reason about liveness properties of TM systems. The framework separates liveness properties of transactions from liveness properties of transactional operations. For example, a TM implementation might satisfy global progress, which requires some correct transaction to commit, and wait-freedom, which requires every correct operation within a transaction to return a response. Our definition of a TM-liveness property conforms to standard general definitions of liveness [2, 21, 25] in the sense that (i) it is a *trace* property [21, 25] (i.e. it is defined in terms of invocations and responses which are external events) and (ii) it allows any *finite* execution [2].

We proved that it is impossible to guarantee both local progress, the strongest TM-liveness property, and opacity in any fault-prone system. There are several ways to circumvent our impossibility result. One way is to weaken safety or TM-liveness property requirements, for example, to require only global progress. There are implementations that ensure opacity and global progress, e.g., OSTM [10]. A second way is to assume that all transactions are static and predefined. That is, when a transaction T starts a TM implementation knows exactly which operations, on which t-variables, will be invoked in T, and the operations invoked in T should be the same in any execution. In that case transactions can be viewed as simple operations and one can apply classical universal construction [14] to ensure local progress. A third way is to assume a fault-free system, i.e. assume that no process can crash or be parasitic. However, it was shown in [20] that even in a fault-free system it is impossible to guarantee opacity and local progress when a TM implementation uses a *direct-update* algorithm and the result can be circumvented only for *deferred-update* algorithms. An algorithm is deferred-update if every transaction that writes a value must invoke a commit request before other transactions can read that value; an algorithm which is not deferred-update is called direct-update. A fourth way is to assume a different system model instead of the multi-threaded programming model. For example, [28] shows a TM implementation that ensures local progress in an asynchronous multicore system model which assumes that a transaction can be executed by different processes and that some process crashes are detectable by the runtime system.

References

[1] Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. ACM Trans. Program. Lang. Syst. 33(1) (2011)

[2] Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21(4) (1985)

[3] Bushkov, V., Guerraoui, R., Kapałka, M.: On the liveness of transactional memory. In: Proceedings of ACM PODC '12 (2012)

[4] Dice, D., Shalev, O., Shavit, N.: Transactional locking ii. In: DISC'06. pp. 194–208. Springer-Verlag (2006)

[5] Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Electron. Notes Theor. Comput. Sci. 259 (2009)

[6] Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects of Computing 25(5), 769–799 (2013)

[7] Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. SIGPLAN Not. 44(6), 155–165 (2009)

[8] Dziuma, D., Fatourou, P., Kanellou, E.: Survey on consistency conditions. Tech. Rep. 439, FORTH-ICS (2013)

[9] Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: ACM PPoPP '08. pp. 237–246 (2008)

[10] Fraser, K.: Practical lock freedom. In: PhD thesis, Cambridge University Computer Laboratory (2003)

[11] Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of ACM PPoPP '08 (2008)

[12] Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Morgan and Claypool (2010)

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

[14] Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1) (1991)

[15] Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of ACM PODC'03 (2003)

[16] Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2) (1993)

[17] Herlihy, M., Shavit, N.: On the nature of progress. In: Proceedings of OPODIS '11. Springer-Verlag (2011)

[18] Imbs, D., de Mendivil, J.R., Raynal, M.: Brief announcement: Virtual world consistency: A new condition for stm systems. In: ACM PODC '09. pp. 280–281 (2009)

[19] Jagannathan, S., Vitek, J., Welc, A., Hosking, A.: A transactional object calculus. Sci. Comput. Program. 57(2) (2005)

[20] Lesani, M., Palsberg, J.: Proving non-opacity. In: Distributed Computing, LNCS, vol. 8205, pp. 106–120. Springer Berlin Heidelberg (2013)

[21] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)

[22] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Practical weak-atomicity semantics for java stm. In: ACM SPAA '08. pp. 314–325 (2008)

[23] Moore, K.F., Grossman, D.: High-level small-step operational semantics for transactions. In: ACM POPL '08. pp. 51–62 (2008)

[24] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4) (1979)

[25] Segala, R., Gawlick, R., Søgaard-Andersen, J., Lynch, N.: Liveness in timed and untimed systems. Inf. Comput. 141(2) (1998)

[26] Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of ACM PODC '95 (1995)

[27] Völzer, H., Varacca, D.: Defining fairness in reactive and concurrent systems. J. ACM 59(3) (2012)

[28] Wamhoff, J.T., Fetzer, C.: The universal transactional memory construction. In: TRANSACT 11. ACM New York, NY, USA, San Jose, CA, USA (June 2011)