Verification of Liveness Properties in Transactional Memories

Victor Bushkov
LPD, I&C, EPFL

Abstract—Transactional memory (TM) is a concurrency control mechanism that avoids common problems associated with conventional locking techniques. The correctness of concurrent programs employing a TM implementation depends on the correctness of the TM implementation. Therefore, it is important to ensure that the implementation satisfies correctness properties such as safety and liveness properties. Safety properties can be ensured by simply aborting operations. Therefore, besides safety properties it is necessary to ensure liveness properties that guarantee progress of threads. Verifying liveness properties of TMs is a difficult task due to the unbounded number of variables and threads. However, if a TM implementation satisfies certain symmetry properties, it is possible to reduce the model checking problem of the implementation to a finite state problem.

Index Terms—formal methods, transactional memory, liveness properties, verification

I. INTRODUCTION

Explicit locking both fine-grained and coarse-grained has many problems including priority inversion, convoying, and difficulty of avoiding deadlocks. To avoid them a concept of transactional memory (TM) was introduced. Transactional memory can be implemented either in hardware [4], in software [1], or a combination of two, this work is focused on software implementations of TMs. Transactional memory provides programmers with easy to use synchronization abstraction called a transaction. A transaction is a finite sequence of machine instructions, executed by a single thread (process), satisfying the following properties. First, transactions appear as if they are executed instantaneously. Second, transactions make changes to shared memory, after a transaction completes, it either commits, making its changes visible to other threads instantaneously, or it aborts, causing its changes to be discarded.

Since a transactional memory implementation can be used by many concurrent applications and the correctness of these applications depends on the correctness of the implementation, it is necessary to make sure that the implementation is correct. A lot of work [3], [6], [5] is focused on safety correctness properties, i.e. on what transactional memory should not do. It is obvious that such transactional memory can be implemented by blocking all operations executed by threads. Therefore, it is important to ensure that besides satisfying safety properties a transactional memory implementation should not prevent threads from making progress, i.e. it should satisfy liveness properties.

Roughly speaking, a liveness property states that some good events will eventually happen. For example a good event can be a program termination, a thread entering its critical section, or an answered request. In the case of transactional memories good events are commit events of transactions. Liveness is a trace set property, and therefore it can be formalized by I/O automata [2]. Formalizing liveness properties by I/O automata allows for verification of liveness properties by checking trace inclusion of an implementation into specification.

In TMs the number of threads and the number of transactional variables (memory locations) can be arbitrary which complicates the problem of modeling transactional memory implementations and specifications. In [3] transactional memory implementations are modeled as infinite sets of automata each corresponding to a particular number of threads and variables. Because sets of automata are infinite, the verification cannot be feasibly solved. However, the problem can be reduced to a finite state problem, if a TM implementation satisfies some structural symmetry properties. Namely, if the symmetry properties are satisfied, then it is sufficient to check trace inclusion of implementation into specification for 2 threads and 2 variables. Most transactional memory implementations like sequential, two-phase locking, dynamic
EDIC RESEARCH PROPOSAL

II. SOFTWARE TRANSACTIONAL MEMORY

Software Transactional Memory can be defined as a generic non-blocking synchronization construct that allows correct sequential objects to be converted automatically into correct concurrent objects. A transaction is defined as a finite sequence of instructions: read-transactional – which reads value from a shared memory, write-transactional – which writes value to a shared memory. Sequence of instructions are applied to a set of memory locations called data set and should satisfy two properties:

- Atomicity – sequence of transactions appears to execute sequentially, i.e. there exists equivalent sequential sequence.
- Serializability – sequential order among non-overlapping transaction is consistent with their real-time order.

An implementation is non-blocking if some processes will terminate successfully after finite number of repeated executions of some transactions.

A. Static Software Transactional Memory

The implementation of the Static Software Transactional Memory proposed by [1] is the following. A transaction makes update to a concurrent object only after a declaration to other transactions of its update intention. This declaration helps other transactions to recognize that some transaction is about to make an update to a concurrent object. Then the declaring transaction is the owner of the corresponding concurrent object. The declaration is done by storing a reference in the concurrent object to its current owner transaction. After making the intended update to the owned concurrent object, a transaction releases its ownership. To ensure atomic updates to concurrent objects, transactions need exclusive ownerships of these objects called cells. At most one transaction may own a cell at any given time. The ownership information of a cell is stored in another corresponding cell called the ownership record. Each memory cell has a distinct associated ownership record. The ownership record either stores a null value (meaning that no transaction owns the corresponding shared memory word currently) or a reference to its owners transaction record. A transaction record stores information about the corresponding transactions software transactional memory accesses. A transaction owns a single transaction record at a time. All the transactions are given shared access to all the existing transaction records, but each transaction record is owned by only one transaction.

If a transaction fails to acquire an ownership (because the memory location is owned by some other transaction at that time), it aborts and releases its already acquired ownerships. If the transaction succeeds in acquiring all the desired ownerships, it atomically changes its state to a committed state, makes its updates, and releases the acquired ownerships. Since all the processes executing a transaction read the same data set vector and will never acquire ownership after the state of the transaction was set to committed, then the following lemma holds.

Lemma 1: The implementation is atomic and serializable.

To avoid livelocks, ownerships are acquired in some global total order of the shared memory word addresses. In order to ensure fault tolerance a non-recursive helping mechanism is used forcing conflicting transactions to help the owner of the ownership record under conflict. In helping, when a transaction detects a conflict with another transaction, the transaction that detected a conflict makes updates of the other conflicting transaction on its behalf. This may lead to recursion and possibly livelocks. Non-recursive helping allows helping up to a specified level only. Total global ordering and helping ensure that the implementation is non-blocking. Thus, the following lemma holds.

Lemma 2: The implementation non-blocking.

During an empirical evaluation the presented implementation of the Static Software Transactional Memory outperformed Herlihy’s method (which is based on copying a data structure to a new block of memory, making changes to the new version, and switching the pointer to the new data structure) and cooperative method (which is based on recursive helping of threads to each other). However it is inferior to blocking lock-based synchronization techniques such as queue-locks.

III. LIVENESS PROPERTIES

While safety properties state that some undesirable events will never happen, liveness properties state that some desirable events will eventually happen. To reason about liveness properties of distributed and concurrent systems an I/O automata framework was introduced [2]. It allows both modeling and verification of different safety and liveness properties of distributed and concurrent systems.

A. Safe and Live Automata

An automaton A is the following tuple:

- \((\text{states}(A), \text{start}(A), \text{sig}(A), \text{steps}(A))\), where
- \(\text{states}(A)\) is a set of states,
- \(\text{start}(A) \subseteq \text{states}(A)\) is a set of start states,
- \(\text{sig}(A) = (\text{ext}(A), \text{int}(A))\) is a signature of consisting of external and internal actions,
- \(\text{steps}(A) \subseteq \text{states}(A) \times \text{acts}(A) \times \text{states}(A)\) is a transition relation, where \(\text{acts}(A) = \text{ext}(A) \cup \text{int}(A)\).

An automaton A can be thought as a safety property stating which actions are not allowed to be executed at some states. Then a liveness property can be described by a subset \(L\) of
the executions of $A$. However, not every subset $L$ of the executions of $A$ can be a liveness property, because some subsets introduce more safety which is already defined by $A$. If some finite execution $\alpha$ of $A$ cannot be extended to some execution of $L$, then $\alpha$ is not allowed in the system described by $(A,L)$, thus, $L$ introduces additional safety property. For $L$ to be a liveness property, every finite execution of $A$ must be extensible to some execution from $L$, formally:

Definition 1: A subset $L$ of the executions of an automaton $A$ is a liveness condition for $A$ iff for every finite execution $\alpha$ of $A$ there exists an execution $\alpha'$ such that $\alpha \cdot \alpha' \in L$.

### B. Safe and Live I/O Automata

Since most distributed and concurrent systems interact with other systems and environment, an I/O automata model is considered. An I/O automaton $A$ is an automaton in which external actions are partitioned into input and output actions by an external action signature $\text{esig}(A) = (\text{in}(A), \text{out}(A))$ and in each state of which every input action is enabled. Actions which are controlled by $A$, i.e. actions which can be disabled by $A$, are called locally controlled $\text{local}(A) = \text{int}(A) \cup \text{out}(A)$.

An I/O automaton which defines some safety property is called a safe I/O automaton. To build complex systems out simpler component the parallel composition is used. The parallel composition can be defined only for compatible I/O automata. Basically, two I/O automata are compatible if each action is an output action of at most one I/O automata and internal action names are unique.

Definition 2: The parallel composition $A_0 \parallel A_1$ of two compatible I/O automata $A_0$ and $A_1$ is the I/O automaton $A$ such that:

- $\text{states}(A) = \text{states}(A_0) \times \text{states}(A_1)$
- $\text{start}(A) = \text{start}(A_0) \times \text{start}(A_1)$
- $\text{out}(A) = \text{out}(A_0) \cup \text{out}(A_1)$
- $\text{in}(A) = (\text{in}(A_0) \cup \text{in}(A_1)) \setminus \text{out}(A)$
- $\text{int}(A) = \text{int}(A_0) \cup \text{int}(A_1)$
- $((s_a, s_b), a, (s'_a, s'_b)) \in \text{steps}(A)$ iff $\forall i \in \{0, 1\}$
  - if $a \in \text{acts}(A_i)$, then $(s_a, i, s'_b) \in \text{steps}(A_i)$
  - if $a \notin \text{acts}(A_i)$, then $s'_a = s'_a$

To define liveness properties over I/O automata we cannot use the same approach as in the Definition 1, because this approach does not capture the fact that system should behave properly independently of it’s environment. For example, consider an I/O automaton $A = (\{s\}, \{s\}, \{(a, b), \emptyset\}, \{(s, a, s), (s, b, s)\})$ with $\text{esig}(A) = \{(a), (b)\}$ and let $L = \{a \alpha a \mid a \text{ contains at least five occurrences of action } a\}$. By the Definition 1, the set $L$ is a liveness condition for $A$. However, the pair $(A, L)$ would not behave correctly if an environment provides less than five actions $a$.

To capture the fact that system should behave correctly independently of its environment the notion of receptiveness is introduced. The interaction between a system and its environment is represented as a two-person game. A system is considered receptive if it can win the game independently of the moves of the environment. To formally define receptiveness we need to define a strategy and an outcome of a strategy.

Definition 3: Let $A$ be an I/O automaton. A strategy on $A$ is a pair of functions $(g, f)$ where $g : \text{execute}^*(A) \times \text{init}(A) \to \text{states}(A)$ and $f : \text{execute}^*(A) \to (\text{local}(A) \times \text{states}(S)) \cap \{\downarrow\}$ such that:

- $g(\alpha, a) = s$ implies that $a \alpha s \in \text{execute}^*(A)$.
- $f(\alpha) = (a, s)$ implies that $a \alpha s \in \text{execute}^*(A)$.

Definition 4: Let $A$ be an I/O automaton and $(g, f)$ a strategy on $A$. Define an environment sequence for $A$ to be any infinite sequence of symbols from $\text{in}(A) \cup \{\lambda\}$ with infinitely many occurrences of $\lambda$. Then define $R(g, f)$, the next-function induced by $(g, f)$, as follows: for any finite execution $\alpha$ of $A$ and any environment sequence $l$ for $A$: $R(g, f)(\alpha, l) = (\alpha a, f)$ if $l = \lambda^0$ and $f(\alpha) = (a, s), R(g, f)(\alpha, l) = (\alpha b, f)$ if $l = \lambda^1$ and $f(\alpha) = (a, s), R(g, f)(\alpha, l) = (\alpha a, f)$ if $l = \alpha l^0$ and $g(\alpha, a) = s$. Then for any finite execution $\alpha$ of $A$ and any environment sequence $l$ for $A$ the outcome sequence of $(g, f)$ given $\alpha$ is the unique infinite sequence $(\alpha^n, l^n)_{n \geq 0}$ such that:

- $(\alpha^0, l^0) = (\alpha, l)$
- $\forall n > 0, (\alpha^n, l^n) = R(g, f)(\alpha^{n-1}, l^{n-1})$

The outcome $O(g, f)(\alpha, l)$ of the strategy $(g, f)$ for given $\alpha$ and $l$ is the execution $\lim_{n \to \infty} \alpha^n$, where $(\alpha^n, l^n)_{n \geq 0}$ is the outcome sequence of $(g, f)$.

Receptiveness requires that there exists a strategy, called a receptive strategy, such that the strategy allows the system to win every game against its environment. Formally, receptiveness is defined in the following way:

Definition 5: Let $A$ be an I/O automaton and $L$ be a subset of its executions. A strategy $(g, f)$ on $A$ is a receptive strategy for $(A, L)$ if for any finite execution $\alpha$ of $A$ and any environment sequence $l$, the outcome $O(g, f)(\alpha, l)$ is an element of $L$. The pair $(A, L)$ is receptive if there exists a receptive strategy.

It can be shown that if $(A, L)$ is a receptive pair, then $L$ is a liveness condition. Thus, a live I/O automaton is a receptive pair $(A, L)$. Now we can define the parallel composition operator for two live I/O automata. The parallel composition of two live I/O automata $(A_1, L_1)$ and $(A_2, L_2)$ is a pair $(A, L)$ such that $A = A_1 \parallel A_2$ and $L = \{\alpha[\text{projection of } \alpha \text{ on actions of } A_1 \text{ and } A_2] \mid L_1 \}$. The parallel composition operator is closed under live I/O automata, thus the following theorem holds:

Theorem 1: If $(A_1, L_1)$ and $(A_2, L_2)$ are live I/O automata, then $(A_1, L_1) \parallel (A_2, L_2)$ is a live I/O automaton.

Receptiveness property guarantees that components do not constrain environments of each other when composed in a parallel composition. For example, consider an I/O automaton $A = (\{s_A\}, \{s_A\}, \{(a, b), \emptyset\}, \{(s, a, s), (s, b, s)\})$ with $\text{esig}(A) = \{(a), (b)\}$ and $B = (\{s_B\}, \{s_B\}, \{(a, b), \emptyset\}, \{(s, a, s), (s, b, s)\})$ with $\text{esig}(B) = \{(a), (b)\}$. Let $L_A = \{\alpha [\exists \lambda : \alpha = \alpha' (a b)^{\infty} \text{ or } \alpha = \alpha' (a)^{\infty}\}$ and $L_B = \{\alpha [\exists \lambda : \alpha = \alpha' (a b b)^{\infty} \text{ or } \alpha = \alpha' (b)^{\infty}\}$. If we consider the following environment sequence $l = b b b b \lambda \ldots$, then we can that the pair $(A, L_A)$ is not receptive. If we consider the following environment sequence $l = a a a a a a a \ldots$, then we can that the pair $(B, L_B)$ is not receptive. If we consider the following environment sequence $l = a a a a a a a \ldots$, thus we can that the pair $(B, L_B)$ is not receptive. If we consider the following environment sequence $l = a a a a a a a \ldots$, then $(C, L_C) = (A, L_A) \parallel (B, L_B)$, then $L_C = \emptyset$, thus, $L_C$ is not a liveness condition.
To check if an implementation satisfies some specification we need to define preorder relations for live I/O automata. For two live I/O automata \((A_1, L_1)\) and \((A_2, L_2)\) the safe and live preorders are defined in the following way:

- **Safe preorder:** \((A_1, L_1) \sqsubseteq_S (A_2, L_2)\) iff \(\text{traces}(A_1) \subseteq \text{traces}(A_2)\);
- **Live preorder:** \((A_1, L_1) \sqsubseteq_L (A_2, L_2)\) iff \(\text{traces}(L_1) \subseteq \text{traces}(L_2)\).

Receptiveness of components guarantees substitutivity of the safe and live preorders for parallel composition which is stated in the following theorem:

**Theorem 2:** Let \((A_i, L_i)\), \((A'_i, L'_i)\) be live I/O automata, where \(i \in \{1, 0\}\), and \(\sqsubseteq_X\) be either the safe or live preorder. If for each \(i\), \((A_i, L_i) \sqsubseteq_L (A'_i, L'_i)\), \((A_0, L_0)\) and \((A_1, L_1)\) are compatible, and \((A'_0, L'_0)\) and \((A'_1, L'_1)\) are compatible, then \((A_0, L_0) \parallel (A_1, L_1) \parallel (A'_0, L'_0) \parallel (A'_1, L'_1)\) is compatible.

This theorem allows to verify the correctness of a system by checking correctness of its components instead of checking correctness of the system as a whole. The live preorder allows to infer correctness of an implementation based on live traces and not only on fair traces (which can be restrictive). Safe preorder guarantees that an implementation is not able to do anything which is not allowed by a specification. On the other hand the live preorder guarantees that an implementation will eventually do something which is required by a specification.

**IV. VERIFYING TRANSACTIONAL MEMORIES**

**A. Modeling Transactional Memories by Automata**

A TM implementation can be expressed through TM algorithms formalism which is in turn expressed by an automaton [3]. A TM algorithm is an automaton \(\mathcal{A} = (Q, q_0, D, \phi, \gamma, \delta)\) consisting of a set of states \(Q\), an initial state \(q_0\), an extended set of commands \(D\) depending on the underlying TM, a conflict function \(\phi\), a pending function \(\gamma\), and a transition relation \(\delta\). Let \(V\) be a set of \(\{1, \ldots, n\}\) of \(k\) variables and \(C = \{\text{commit}\} \cup \{(\text{read}, \text{write}) \times V\}\) be the set of commands on variables \(V\) and \(C = C \cup \{\text{abort}\}\). A command in a TM algorithm is executed as a sequence of extended commands, all of which execute atomically. Thus, the extended commands \(D\) include the set \(C\) of commands, as well as TM specific additional commands. For instance, a TM may require that a thread locks a variable before writing to the variable. Let \(T = \{1, \ldots, n\}\) be the set of threads, \(S = \hat{C} \times T\) be the set of states, and \(S = C \times T\). The conflict function \(\phi: Q \times S \rightarrow \{\text{true}, \text{false}\}\) captures the statements in a state, when the TM algorithm may consult a contention manager for a decision. That is, if \(\phi(q, (c, t)) = \text{true}\), then this implies that the TM algorithm can nondeterministically choose to abort the thread \(t\) in state \(q\), if the command \(c\) is issued. The pending function \(\gamma: \mathcal{A} \times T \rightarrow C \cup \{\bot\}\) represents the pending command of a thread in a state, and ensures that if a thread has not finished the execution of all extended commands corresponding to a particular command, then no other command is executed by the thread. The transition relation is a subset of the form \(\delta \subseteq Q \times C \times S_D \times \text{Resp} \times Q\), where \(S_D = (S \cup \{\text{abort}\}) \times T\) and \(\text{Resp} = \{\bot, 1, 0\}\) is the set of responses. The TM algorithm makes a transition according to the transition relation, and gives back to the program a response. The response is \(\bot\) if the TM algorithm needs additional steps to complete the command, \(0\) if the TM algorithm needs to abort the transaction of the scheduled thread, and \(1\) if the TM algorithm has completed the command. A TM algorithm interacts with a scheduler. The scheduler chooses the next thread to be executed. A command of the chosen thread is given to the TM algorithm.

Every TM implementation can be represented by the proposed framework, as an example, [3] focuses on the following TMs. The sequential TM is the simplest implementation which executes transactions only sequentially. The two-phase locking TM uses locks, every transaction locks the variables it reads or writes before accessing them, and releases all acquired locks during the commit. A shared lock is acquired for reading, and an exclusive lock is acquired for writing. In dynamic software transactional memory (DSTM) [8] transactions require ownership of variables only for writing. The readers are not visible to the writers. Upon reading, a transaction validates its read set. Transactional locking 2 [7] uses a global version clock. Every transaction starts by reading the current value of the clock and storing it. Then, on every read or write, the version of the particular memory location is compared to the stored version; and, if it’s greater, the transaction is aborted. During commit, all write locations are locked, and version numbers of all read and write locations are re-checked. Finally, the global version clock is incremented, new write values from the log are written back to memory and stamped with the new clock version.

Usually most TMs employ a separate module, called a contention manager, to enhance liveness. A contention manager resolves conflicts on the basis of the past behavior of the transactions. It is modeled as tuple \((P, p_i, \delta_{cm})\), where \(P\) is a set of states and \(p_i\) is the initial state. The transition relation \(\delta_{cm} \subseteq P \times D \times P\) restricts behavior of TM algorithms by blocking some transitions at states \(q\) such that \(\phi(q, (c, t)) = \text{true}\). For example, the aggressive contention manager does not allow a transaction to abort itself in case of a conflict. While the polite contention manager always requires a transaction to abort in case of conflict.

**B. Verifying Safety Properties**

To define safety properties of transactional memories we need first to formally define the notion of a conflict. Statement \(s_1\) of transaction \(x\) and statement \(s_2\) of transaction \(y\) conflict in some word \(w\) if (i) \(s_1\) is a global read of some variable \(v\), and \(s_2\) is a commit, and \(y\) writes to \(v\), or (ii) \(s_1\) and \(s_2\) are both commits, and \(x\) and \(y\) write to some variable \(v\). A word \(w\) is strictly equivalent to a word \(w'\) if for every thread \(t\) their corresponding projections on \(t\) are equal and (i) for every \(s_i, s_j\) if \(s_i\) and \(s_j\) conflict and \(i < j\), then \(s_i\) occurs before \(s_j\) and (ii) the real time order of transactions is preserved. The safety property strict serializability is defined as the set of words \(w\) such that there exists a sequential word \(w'\), where \(w'\) is strictly equivalent to \(\text{comp}(w)\) and \(\text{comp}(w)\) is derived from \(w\) by eliminating aborting transactions. Opacity is defined as the set of words \(w\) such that there exists a sequential word \(w'\), where \(w'\) is strictly equivalent to \(w\).
To verify that TM implementation satisfies a safety property we need to verify that it satisfies the property for any number of threads and any number of variables. Because modeling contention managers is not a feasible option the safety of a TM should not depend on the choice of the contention manager, i.e., a TM should be safe with any contention manager. When the TM algorithm is used with a contention manager, the transition should exist in the transition relation of the TM algorithm and that of the contention manager. Thus, it is sufficient to prove the safety of a TM without a contention manager, in order to show that the TM using any contention manager is safe. The following four structural properties allow to reduce the verification problem to the case with two threads and two variables.

- **P1** Transaction projection. Aborting and unfinished transactions can influence other transactions only by forcing them to abort. In other words, removing all aborting transactions and some of the unfinished transactions do not change the response of the TM to the remaining statements.
- **P2** Thread symmetry. For non-overlapping transactions the TM cannot identify the thread executing the transaction.
- **P3** Variable projection. If a transaction can commit, then removing all statements that involve some particular variables does not cause the transaction to abort.
- **P4** Monotonicity. If a word is allowed by the TM, then more sequential forms of the word are also allowed.

**Theorem 3:** If a TM $M$ ensures (2, 2) strict serializability (resp. (2, 1) opacity) and satisfies the properties P1, P2, P3, and P4 for strict serializability (resp. opacity), then $M$ ensures strict serializability (resp. opacity).

Thus, to check if a TM is safe we need to check if it satisfies a safe specification for two threads and two variables. A safe specification can either deterministic or non-deterministic. In the case of a non-deterministic specification a transaction nondeterministically guesses a serialization point during its lifetime. A branch of the nondeterministic TM specification corresponds to a specific serialization choice of the transactions. Depending on the choice the TM specification checks upon the commit of a transaction, whether the commit can be executed, or the transaction needs to abort. In the case of a deterministic specification we have to consider all possible serialization orders at the same time, which complicates the specification. Using safe specifications it was verified in [3] that sequential, two-phase locking, dynamical, and transactional locking 2 TMs satisfy safety specification.

C. Verifying Liveness Properties

Liveness of a TM may depend on a contention manager, because the decision of a contention manager may require a thread to wait for an arbitrarily long period of time, or may require a thread to abort any conflicting transaction. Therefore, liveness properties are verified in conjunction with a specific contention manager.

The main liveness properties are obstruction freedom, livelock freedom, and wait freedom. Obstruction freedom requires that if a transaction executes in isolation, then it eventually commits. Livelock freedom requires some transaction to eventually commit. Wait freedom requires every transaction to eventually commit. Wait freedom implies livelock freedom and livelock freedom implies obstruction freedom.

To verify that a TM satisfies some properties the notion of loop is introduced. A loop $l$ in a TM algorithm $M$ is a finite sequence of transitions $(q_0, c_0, q_0, r_0, q_0') \ldots (q_m, c_m, q_m, r_m, q_m')$ such that $q_0 = q_m$. Then a TM defined by a TM algorithm $A$ ensures obstruction freedom iff there is no loop $l$ in $A$ such that all statements in $l$ are from the same thread, and $l$ contains no commit, and $l$ contains an abort. A TM ensures livelock freedom iff there is no loop $l$ in $A$ such $l$ contains no commit, and every thread that has a statement in $l$, has an abort in $l$.

To verify liveness properties of TMs by reducing the verification problem to the verification problem for two threads and two variables the following properties are used:

- **P5** Transaction projection. A thread running in isolation (no interleaved step from other threads) shall abort repeatedly only if it conflicts with some unfinished transaction.
- **P6** Variable projection. A thread running in isolation shall abort repeatedly only if some commands corresponding to some variables are not allowed.

**Theorem 4:** If a TM $M$ ensures (2, 1) obstruction freedom and satisfies the properties P5 and P6, then $M$ ensures obstruction freedom.

Thus, to check if a TM algorithms $A$ (which is a composition of another TM algorithm and some contention manager) satisfies obstruction freedom, it is sufficient to find a loop $l$ in the transition system of $A$ for two threads and two variables such that all statements in $l$ are from the same thread, and $l$ has no commit, and $l$ has an abort. If such loop exists, then $A$ does not ensure obstruction freedom. Likewise, to check if a TM algorithms $A$ satisfies livelock freedom, it is sufficient to find a loop $l$ in the transition system of $A$ for two threads and two variables such that there is no commit in $l$, and every thread that has a statement in $l$, has an abort in $l$. If such loop exists, then $A$ does not ensure livelock freedom. The results of [3] show that DSTM with the aggressive contention manager obstruction freedom but does not ensure livelock freedom. The sequential TM and two-phase locking TM (without a contention manager) do not ensure obstruction freedom. TL2 with the polite contention manager does not ensure obstruction freedom.

V. Conclusion and Research Proposal

Software transactional memory is a promising synchronization mechanisms which aims to simplify concurrent programming [1]. If the TM is implemented correctly the programmer is less likely to introduce concurrency bugs in the code than if he or she had to handle locks explicitly. To ensure that TM is implemented correctly, it is necessary to ensure that the implementation never executes undesirable events. But this can be ensured simply by blocking all operations, which does not guarantee any progress. Therefore, besides ensuring safety properties it is necessary to ensure liveness properties. To formally reason about liveness properties of concurrent and
distributed systems an I/O automata framework was introduced in [2]. This framework defines the correctness of implementations independently of the inputs from the environment. In [3] another automata framework is used in verification of liveness properties of TM implementations. The framework allows to reduce the problem of verification of TM implementations from an infinite state problem (due to the unbounded number of threads and variables) to a finite state problem through the reduction theorem. The reduction theorem states that if a TM implementation enjoys some symmetry properties, then it is sufficient to verify that the implementation (in conjunction with a contention manager) ensures a liveness property for two threads and two variables to verify that the implementation ensures the liveness property for any number of threads and any number of variables.

The reduction theorem can be applied only if the symmetry properties hold. However, checking that a software transactional memory implementation satisfies the symmetry properties is done manually. Therefore, before model checking the implementation one has to manually verify that the implementation satisfies the symmetry properties. To make the solution to the TM verification problem fully automatic we propose to design techniques for automatically checking that a TM implementation satisfies the symmetry properties.

The automata framework proposed by [3] allows verification of a TM implementation as a whole system. While the I/O automata framework proposed by [2] allows for verification of a complex system by checking that all components of the system satisfy desired correctness properties. Therefore we propose to designing an I/O automata framework in the spirit of [2] for transactional memories which would allow to reason about TMs as complex systems composed of simpler components (for example reasoning about the correctness of individual memory locations). To do this, however, a composition operator needs to be defined which expresses a TM implementation as a composition of simpler parts. Also, I/O automata framework would allow to reason about liveness properties of TM implementations independently of their environment (a concurrent program employing the implementation). Another possible use of the framework is reasoning about the correctness of nested transactions.

In the case of transactions with priorities it is necessary to ensure that a transaction with the highest priority is not less progressive than transactions with lower priorities. Therefore, a new framework for reasoning about liveness properties should also allow reasoning when transactions (or threads) are ordered according to their priority.

REFERENCES