Verifying Safety and Liveness for the FlexTM Hybrid Transactional Memory

Parosh Abdulla*, Sandhya Dwarkadas†, Ahmed Rezine‡, Arrvindh Shriraman§ and Yunyun Zhu*,

*Uppsala University, Sweden, Email: {parosh,yunyun.zhu}@it.uu.se

†University of Rochester, U.S.A., Email: sandhya@cs.rochester.edu

‡Linköping University, Sweden, Email: ahmed.rezine@liu.se

§Simon Fraser University, Canada, Email: ashriram@cs.sfu.ca

***Abstract*—We consider the verification of safety (strict serializability and abort consistency) and liveness (obstruction and livelock freedom) for the hybrid transactional memory framework FLEXTM. This framework allows for flexible implementations of transactional memories based on an adaptation of the MESI coherence protocol. FLEXTM allows for both eager and lazy conflict resolution strategies. Like in the case of Software Transactional Memories, the verification problem is not trivial as the number of concurrent transactions, their size, and the number of accessed shared variables cannot be a priori bounded. This complexity is exacerbated by aspects that are specific to hardware and hybrid transactional memories. Our work takes into account intricate behaviours such as cache line based conflict detection, false sharing, invisible reads or non-transactional instructions. We carry out the first automatic verification of a hybrid transactional memory and establish, by adopting a small model approach, challenging properties such as strict serializability, abort consistency, and obstruction freedom for both an eager and a lazy conflict resolution strategies. We also detect an example that refutes livelock freedom. To achieve this, our prototype tool makes use of the latest antichain based techniques to handle systems with tens of thousands of states.**

I. INTRODUCTION

Multicore processors are often presented as the only viable solution to achieve more performance in modern computers. This entails writing concurrent code that takes advantage of the parallelism available on these machines. Traditional concurrent programming, especially based on lock synchronization, can easily become a daunting and error prone task. In this context, the Transactional Memory (TM) approach takes advantage of the offered parallelism while giving the programmer the illusion of atomicity. More concretely, the programmer describes the blocks that are to be executed atomically. Each such block (i.e., a transaction) boils down to a finite sequence of loads and stores. The sequence ends with a commit instruction. A successful transaction takes effect and is said to be committed. Otherwise, the transaction is aborted and may be restarted if needed. The concurrent transactions run on top of an underlying Transactional Memory (hardware [1]–[5], software [6]–[10], or hybrid [11]–[14]) infrastructure that maintains the atomicity illusion.

In this paper, we consider for the first time the problem of automatically and formally verifying a hybrid transactional memory. More concretely, we propose parameterized models (i.e., infinite models with a size that cannot be a priori bounded) in order to capture the intricate behaviors of the FLEXTM hybrid transactional memory [15]. Some of this complexity is shared with software transactional memories, like the arbitrary number of concurrent transactions, their size, the number of shared variables, the granularity of the global operations (like commit) or the handling of non-transactional instructions. Other sources of complexity are exclusive to hardware and to hybrid transactional memories. Aspects like cache line based conflict detection and tracking mechanisms, false sharing and invisible reads have not been taken into account before when verifying properties such as the *abort consistency* of a transactional memory. In fact, the models we propose are so rich that they easily require tens of thousands of states even to represent small instances of the FLEXTM transactional memory. We check against these models the validity of the four challenging properties of *strict serializability*, *abort consistency*, *obstruction freedom* and *livelock freedom* [16]–[18].

Intuitively, strict serializability is a safety property that requires that all transactions appear to take effect *sequentially* (i.e., one after the other) with no interleavings, and such that the order of non-interleaving transactions is preserved. Abort consistency further requires that even aborted transactions observe consistent variables values. On the other hand, *obstruction freedom* and *livelock freedom* are liveness properties that ensure progress. *Obstruction freedom* roughly requires that a transaction in isolation will eventually successfully commit, and *livelock freedom* requires that if several transactions are simultaneously active, then at least one of them will succeed.

Several works have considered the parameterized verification of coherence in cache protocols [19]–[21]. The properties we consider in this work are more complex as they combine several degrees of infiniteness (number of concurrent transactions, their size, and the number of involved shared variables). In addition, and unlike previous work on the verification of software transactional memory [22], we need to take into account behaviours and mechanisms that can only be found in hybrid or hardware transactional memories (false sharing, invisible reads and non-atomic commits with cache line based conflict detection and tracking).

We make the following contributions in this paper.

- We propose two fine grained models for the FLEXTM

978-3-9815370-0-0 / DATE13 / © 2013 EDAA

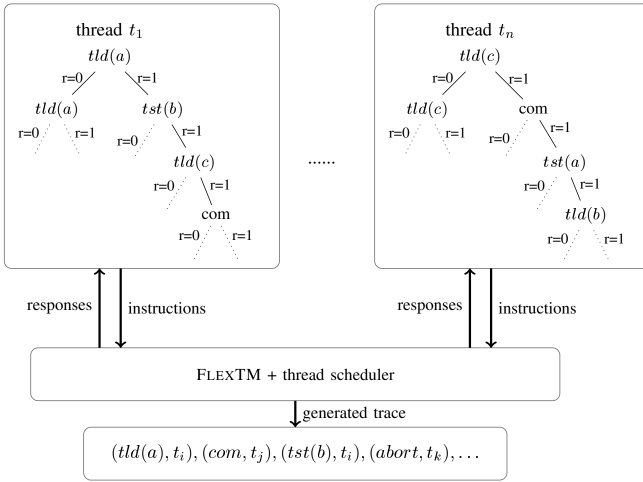


Fig. 1. T concurrent threads access V shared variables by running sequences of (transactional) instructions on top of the FLEXTM hybrid transactional memory. Threads are modelled as binary trees to capture the control flow and the possibilities of abort.

hybrid transactional memory [15]. The models differ in the adopted conflict resolution strategies (i.e., eager and lazy) and take into account the non transactional code in the programs.

- We build these models and obtain automata with tens of thousands of states. We modify them in order to allow write-write conflicts. This is a commonly allowed conflict in software transactional memories. Our prototype tool automatically captures a non serializable execution due to invisible reads and false sharing. The aspects are exclusive to hardware implementations and had to be taken into account, for the first time, when automatically proving strict serializability.
- We state that the obtained models do in fact exhibit symmetry properties introduced in [22] allowing us to make use of a small model theorem for which they propose a most general description for strict serializability and for abort consistency. The properties ensure that it is enough to check the considered properties on models involving two threads and two variables.
- We establish obstruction freedom and refute livelock freedom by analyzing the obtained models.
- We prove strict serializability and abort consistency by language inclusion using recent non trivial automata based techniques [23], [24] that can establish (or refute) in seconds language inclusion for automata with tens of thousands of states.

In the rest of the paper, we introduce the FLEXTM framework and describe its lazy (and most involved) mode in Section II. We formally define in Section III the generated transactions and formally prove the required properties in Section IV. We report on the size of the obtained models, on the used techniques and on the results in Section V. We conclude in Section VI.

II. The FlexTM Transactional Memory

The FLEXTM framework [15] allows for two conflict detection modes, either lazy or eager. In eager mode, a conflict manager is called as soon as a conflict is detected. The conflict manager is then free to decide which transaction to abort. When a transaction reaches its commit point, it can commit if it was not aborted by another transaction. In lazy mode, the hardware keeps track of the conflicts and it is when the transaction tries to commit that it aborts, one by one, the conflicting transactions. This takes place non-atomically in software and can result in races among the competing transactions. In our work, we modeled both modes. To simplify the presentation, and unless otherwise stated, we will refer to the lazy and more involved mode each time we mention FLEXTM.

A. *Threads and programs*

Adopting the approach of [22], an arbitrary but fixed number of threads are assumed to share an arbitrary number of variables and run concurrently, one per processor, on a multiprocessor machine. We write T to mean the set of concurrent threads and V for the set of variables they share. For the sake of clarity, we restrict the presentation in this section to the case where threads only run transactional instructions. We explain how we deal with non-transactional instructions at the end of the present section. As depicted in Fig.1, each thread in T issues transactional loads ($tld(v)$), transactional stores ($tst(v)$), or commit instructions (com); i.e., instructions in $C = \{tld(v), tst(v)\}_{v \in V} \cup \{com\}$. A statement is an element of $S = C \times T$. We write $\hat{C}$ and $\hat{S}$ to mean $C \cup \{abort\}$ and $\hat{C} \times T$ respectively. A binary tree $\theta : \{0,1\}^* \to C$ is used to capture the sequences of instructions issued by some thread in T. Intuitively, the control flow is already incorporated in the possibly infinite trees, and a thread sends an instruction to the FLEXTM transaction memory which either accepts it (i.e., returns 1 and the thread proceeds with the right child in the tree) or rejects it (i.e., returns 0 and the thread proceeds with the left child in the tree). We write Θ to mean the set of unrolled threads, i.e., of binary trees over C. A program p is a tuple $(\theta[1], \theta[2], \ldots)$ of unrolled threads, one for each thread $t_1, t_2, \ldots$ in T. P_k^n is the set of programs for n threads and k variables.

B. *The* FLEXTM *transactional memory*

FLEXTM builds on the inherent versioning capabilities of the MESI [25] cache coherence protocol. Schematically, each processor has a private L1 cache and shares the L2 cache with other processors. Given a variable v in V, we write $\$(v)$ to mean the cache line associated with v, and $\$(V)$ to mean the set of all cache lines. We also write $v' \in \$(v)$ to mean that v and v' share the same cache line. When a processor accesses a variable v for read or write, it atomically brings the whole line $\$(v)$ to its cache. This is known as *invisible reads* and amounts to simultaneously reading all variables associated to $\$(v)$. In the descriptions that follow, and given a thread t, we use $\left(\cdot_{v' \in \$(v)}(tld(v'), t)\right)$ to mean a sequence

$(tld(v_1), t) \cdot (tld(v_2), t) \cdots (tld(v_m), t)$ of instructions where thread t reads, in some fixed arbitrary order, all variables in $\$(v)$. For example, we write $\left(\cdot_{v_i \in \$(v)} (tld(v_i), t)\right) \cdot (tst(v), t)$ to mean a sequence of instructions where t reads all variables in $\$(v)$ and then writes to v itself. FLEXTM handles in software the management of the lines that do not fit in the cache. We therefore abstract away from this aspect and uniformly handle all cache lines in the same way.

FLEXTM induces a transition system (Q, Δ, s_{init}). A state s in Q is a tuple $(status, sigs, cst, cst_\nu)$ where:

- $status : T \to \{idle, active, check, abort\}$ keeps track of the status of the current transaction of each thread in T.
- $sigs$ is a pair (r_{sig}, w_{sig}) of mappings $T \to 2^{\$(V)}$ that keep track, for a thread t, of the cache lines of the variables that are read (in $r_{sig}(t)$) or written (in $w_{sig}(t)$) by its current transaction.
- cst is a tuple (rw, wr, ww) of three mappings $T \to 2^T$ that associate to each thread the set of threads with transactions that have a cache line based (read-write, write-read, or write-write) conflict with its current transaction.
- cst_ν is a pair of mappings used during the nonatomic commit phase, they hold copies of wr and ww.

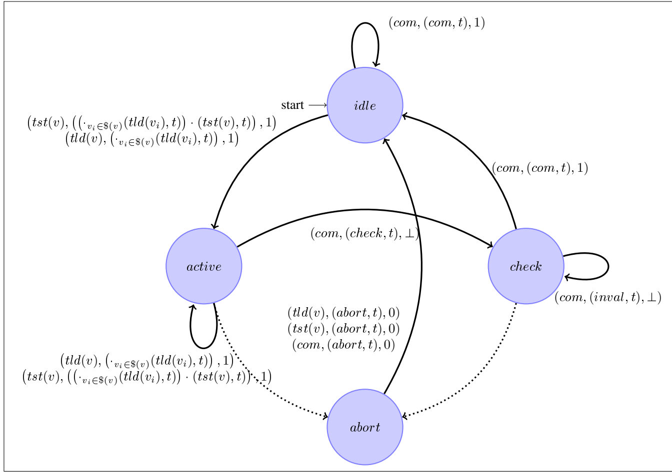
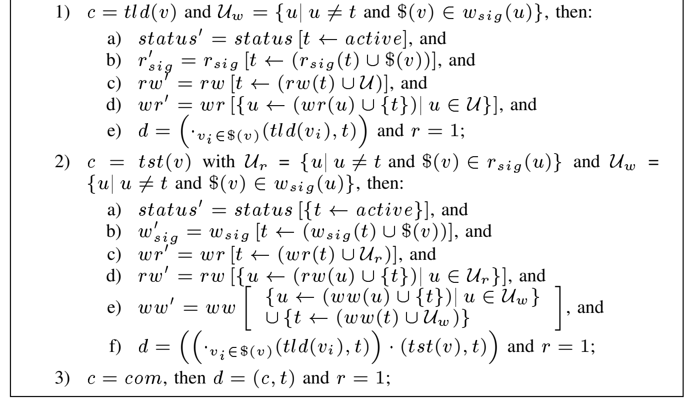


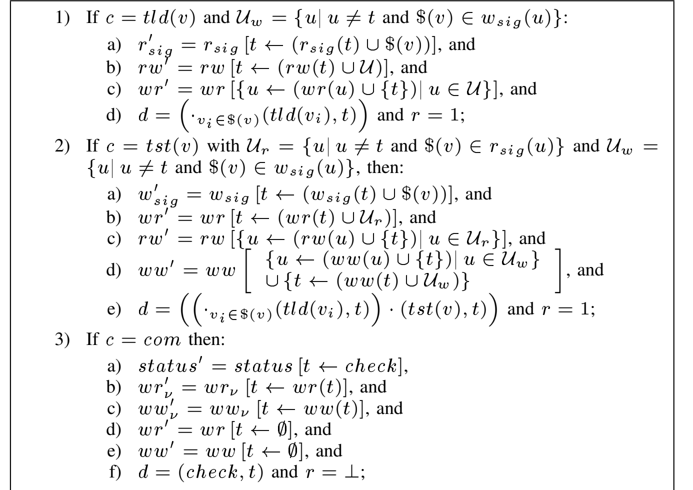
Fig. 2. Possible status changes for a thread t in the lazy mode of FLEXTM.

The initial state s_{init} maps all threads to the $idle$ status, and all signatures and tables to the empty ones. We let Q be the set of states of the FlexTM. Fig.2 describes the possible status changes for a thread in FLEXTM. A transition in Δ is a tuple (s, c, d, r, s'), where $s, s' \in Q$, $c \in C$ and d in $\{(c_1, t) \cdots (c_m, t) | \forall i : 1 \leq i \leq m : c_i \in C \text{ and } t \in T\} \cup (\{abort, check, inval\} \times T)$ and $r \in \{0, 1, \bot\}$. We write threadOf(d) to mean the thread issuing the instructions in d. Intuitively, the thread t issues an instruction c which has as effect that the (sequence of) statement(s) d is executed by thread threadOf(d). FLEXTM responds to the instruction using r to state whether c has been accepted ($r = 1$ and the thread can proceed with the following instruction in the thread, or with a new transaction if c was a successful commit), rejected and the current transaction is aborted ($r = 0$ and $d = (abort, t)$), or



1) $c = tld(v)$ and $\mathcal{U}_w = \{u | u \neq t \text{ and } \$(v) \in w_{sig}(u)\}$, then:
   a) $status' = status\,[t \leftarrow active]$, and
   b) $r'_{sig} = r_{sig}\,[t \leftarrow (r_{sig}(t) \cup \$(v))]$, and
   c) $rw' = rw\,[t \leftarrow (rw(t) \cup \mathcal{U})]$, and
   d) $wr' = wr\,[\{u \leftarrow (wr(u) \cup \{t\}) | u \in \mathcal{U}\}]$, and
   e) $d = \left(\cdot_{v_i \in \$(v)} (tld(v_i), t)\right)$ and $r = 1$;
2) $c = tst(v)$ with $\mathcal{U}_r = \{u | u \neq t \text{ and } \$(v) \in r_{sig}(u)\}$ and $\mathcal{U}_w = \{u | u \neq t \text{ and } \$(v) \in w_{sig}(u)\}$, then:
   a) $status' = status\,[\{t \leftarrow active\}]$, and
   b) $w'_{sig} = w_{sig}\,[t \leftarrow (w_{sig}(t) \cup \$(v))]$, and
   c) $wr' = wr\,[t \leftarrow (wr(t) \cup \mathcal{U}_r)]$, and
   d) $rw' = rw\,[\{u \leftarrow (rw(u) \cup \{t\}) | u \in \mathcal{U}_r\}]$, and
   e) $ww' = ww \left[\begin{array}{c} \{u \leftarrow (ww(u) \cup \{t\}) | u \in \mathcal{U}_w\} \\ \cup \{t \leftarrow (ww(t) \cup \mathcal{U}_w)\} \end{array} \right]$, and
   f) $d = \left(\left(\cdot_{v_i \in \$(v)} (tld(v_i), t)\right) \cdot (tst(v), t)\right)$ and $r = 1$;
3) $c = com$, then $d = (c, t)$ and $r = 1$;

Fig. 3. Thread t with $idle$ status issues an instruction c



1) If $c = tld(v)$ and $\mathcal{U}_w = \{u | u \neq t \text{ and } \$(v) \in w_{sig}(u)\}$:
   a) $r'_{sig} = r_{sig}\,[t \leftarrow (r_{sig}(t) \cup \$(v))]$, and
   b) $rw' = rw\,[t \leftarrow (rw(t) \cup \mathcal{U})]$, and
   c) $wr' = wr\,[\{u \leftarrow (wr(u) \cup \{t\}) | u \in \mathcal{U}\}]$, and
   d) $d = \left(\cdot_{v_i \in \$(v)} (tld(v_i), t)\right)$ and $r = 1$;
2) If $c = tst(v)$ with $\mathcal{U}_r = \{u | u \neq t \text{ and } \$(v) \in r_{sig}(u)\}$ and $\mathcal{U}_w = \{u | u \neq t \text{ and } \$(v) \in w_{sig}(u)\}$, then:
   a) $w'_{sig} = w_{sig}\,[t \leftarrow (w_{sig}(t) \cup \$(v))]$, and
   b) $wr' = wr\,[t \leftarrow (wr(t) \cup \mathcal{U}_r)]$, and
   c) $rw' = rw\,[\{u \leftarrow (rw(u) \cup \{t\}) | u \in \mathcal{U}_r\}]$, and
   d) $ww' = ww \left[\begin{array}{c} \{u \leftarrow (ww(u) \cup \{t\}) | u \in \mathcal{U}_w\} \\ \cup \{t \leftarrow (ww(t) \cup \mathcal{U}_w)\} \end{array} \right]$, and
   e) $d = \left(\left(\cdot_{v_i \in \$(v)} (tld(v_i), t)\right) \cdot (tst(v), t)\right)$ and $r = 1$;
3) If $c = com$ then:
   a) $status' = status\,[t \leftarrow check]$,
   b) $wr'_\nu = wr_\nu\,[t \leftarrow wr(t)]$, and
   c) $ww'_\nu = ww_\nu\,[t \leftarrow ww(t)]$, and
   d) $wr' = wr\,[t \leftarrow \emptyset]$, and
   e) $ww' = ww\,[t \leftarrow \emptyset]$, and
   f) $d = (check, t)$ and $r = \bot$;

Fig. 4. Thread t with $active$ status issues an instruction c

the thread has to wait for FLEXTM to respond ($r = \bot$). The third case occurs when c is a commit, it results in transitions that involve some instructions that are internal to FLEXTM, namely to start the commit phase (with $check$) or to abort conflicting transactions (with $inval$). In this case $r = \bot$ and $d \in \{check, inval\} \times T$. If a read (resp. write) to variable v is accepted by FLEXTM ($r = 1$), then we take into account the entailed invisible reads with $d = \left(\cdot_{v_i \in \$(v)} (tld(v_i), t)\right)$ (resp. $d = \left(\cdot_{v_i \in \$(v)} (tld(v_i), t) \cdot (tst(v))\right)$). The transitions Δ are depicted in Fig.3, Fig.4, Fig.5 and Fig.6. In these figures, we write $f[x \leftarrow b]$ to mean the mapping g with the same domain as f and such that $g(y) = f(y)$ for all $y \neq x$ and $g(x) = b$. In addition, we let $s = \left(status, (r_{sig}, w_{sig}), (rw, wr, ww), (wr_\nu, ww_\nu)\right)$ and $s' = \left(status', (r_{sig}', w'_{sig}), (rw', wr', ww'), (wr'_\nu, ww'_\nu)\right)$

For example, Fig.5 describes a thread t running the non-atomic commit procedure. In this description, t aborts conflicting threads in an $active$ or a $check$ status (case 1). If there are no more conflicting threads and no new conflicts have been added since last copying the conflicts in wr and ww, then the transaction is committed and the status of the new transaction is set to $idle$ (case 3). Otherwise, a new round that takes into account the new conflicts is started (case 2). In

1) If there is a thread $u \neq t$ with u in $(ww_\nu(t) \cup wr_\nu(t))$, then:
   a) $ww'_\nu = ww_\nu [t \leftarrow ww_\nu(t) \setminus \{u\}]$
   b) $wr'_\nu = wr_\nu [t \leftarrow wr_\nu(t) \setminus \{u\}]$,
   c) if $status(u) \in \{active, check\}$ then
      i) $status' = status [u \leftarrow abort]$
      ii) $d = (inval, u)$ and $r = \perp$;
2) If $(ww_\nu(t) \cup wr_\nu(t)) = \emptyset$ but $(ww(t) \cup wr(t)) \neq \emptyset$, then
   a) $wr'_\nu = wr_\nu [t \leftarrow wr]$, and
   b) $ww'_\nu = ww_\nu [t \leftarrow ww]$, and
   c) $wr' = wr [t \leftarrow \emptyset]$, and
   d) $ww' = ww [t \leftarrow \emptyset]$, and
   e) $d = (check, t)$ and $r = \perp$;
3) If $(ww_\nu(t) \cup wr_\nu(t) \cup ww(t) \cup wr(t)) = \emptyset$, then
   a) $status' = status [t \leftarrow idle]$, $d = (com, t)$ and $r = 1$;

Fig. 5. Thread t runs the non-atomic commit, i.e., $status(t) = check$.

1) $status' = status [t \leftarrow idle]$,
2) $S' = S [t \leftarrow \emptyset]$ for $S \in \{w_{sig}, r_{sig}, rw, wr, ww, wr_\nu, ww_\nu\}$,
3) $d = (abort, t)$ and $r = 0$;

Fig. 6. The transaction of thread t has been aborted, i.e. $status(t) = abort$.

addition, the transaction run by thread t can be aborted by concurrent transactions running on other processors (cases 1 during the commit of conflicting transactions in Fig.5).

C. Generated words and transaction.

A run of a program $p = (\theta[1], \ldots, \theta[n])$ in P_k^n with a scheduler $\sigma : \mathbb{N} \to \{i | 1 \leq i \leq n\}$ on a FlexTM is defined as follows. A run is a sequence $\rho = (s_0, l_0) \xrightarrow{\delta_0} (s_1, l_1) \xrightarrow{\delta_1} \ldots$ where $s_i \in Q$, $l_i = (l_i[1], \ldots, l_i[n]) \in (\{0,1\}^*)^n$, with $s_0 = s_{init}$ and $l_0 = (\epsilon, \ldots, \epsilon)$, and such that for all $j \geq 0$, $\delta_j = (s_j, c_j, d_j, r_j, s_{j+1})$ in Δ with: $t_{\sigma(j)} = \text{threadOf}(d_j)$, $c_j = \theta[\sigma(j)](l_j[\sigma(j)])$ and for all $t \in T$, if $t \neq t_{\sigma(j)}$ or $r_j = \perp$ then $l_{j+1}[\sigma(j)] = l_j[\sigma(j)]$, otherwise $l_{j+1}[\sigma(j)] = l_j[\sigma(j)] \cdot r_j$. We write $\rho_{|\hat{S}}$ to mean the projection of the sequence $d_0 \cdot d_1 \cdots$ on $\hat{S}^*$. A word $w \in \hat{S}^*$ is said to be generated by a program p, and write $w \in \mathcal{L}(p)$, if $w = \rho_{|\hat{S}}$ for some run ρ of p with respect to some scheduler.

D. Transactions

Given a finite word w in $\hat{S}^*$, we project on a thread t and obtain a sequence $w_{|t}$. We define terminating statements to be the statements in $\{(com, t), (abort, t)\}$ in addition to the last statement in $w_{|t}$. An initiating statement is either the first statement of $w_{|t}$ or a statement following a terminating statement. A transaction of thread t in a word $w \in \mathcal{L}(p)$ is any contiguous subsequence of $w_{|t}$ that starts with an initiating statement and finishes with a terminating statement, without any terminating statement in the middle. We let $Tx_t(w)$ be the set of transactions of thread t in the word w. A statement s in a transaction x is a global read of a variable v if there are no statements in x that precede s and that write to v. Given a transaction $x \in Tx_t(w)$, x is said to be committing if $(com, t) \in x$ (i.e. if the statement (com, t) appears in x), aborting if $(abort, t) \in Tx_t(w)$, and pending otherwise. Given a word w, we write $com(w)$ to mean the largest subsequence of w that only contains statements of committing transactions in w. The projection of a word w on a set X of transactions in w is the largest subsequence of w that only includes statements belonging to a transaction in X. The variable projection of a word w on a set V' of variables is the longest subsequence of w that only contains commit or abort statements, or statements that read or write variables in V'. Given two transactions x and y in a word w, we write $x <_w y$ to mean that the initiating statement of y appears, in w, after the terminating statement of x. A word is *sequential* if for every pair of transactions x, y in the word, either $x <_w y$ or $y <_w x$.

1) $c = ld(v)$, then:
   a) $d = \left(\cdot_{v_i \in \$(v)}(tld(v_i), t) \cdot (com, t)\right)$ and $r = 1$;
2) $c = st(v)$ with $\mathcal{U} = \{u | u \neq t, \$(v) \in w_{sig}(u) \cup r_{sig}(u)\}$, then:
   a) $status' = status [\{u \leftarrow aborted | u \text{ in } \mathcal{U}\}]$, and
   b) $d = \left(\left(\cdot_{v_i \in \$(v)}(tld(v_i), t)\right) \cdot (tst(v), t) \cdot (com, t)\right)$, $r = 1$;

Fig. 7. Thread t with $idle$ status issues a non-transactional instruction c

E. Non-transactional instructions

In reality, FLEXTM also allows non-transactional instructions in programs. A non-transactional instruction is either a read $ld(v)$ or a write $st(v)$ of some variable $v \in V$. FLEXTM treats any non-transactional read (resp. write) instruction c issued by a thread t as a transactional read (resp. write) instruction if it appears within a transaction, i.e. there is a non-terminating transactional instruction c' issued by t that precedes c and such that there is no terminating transaction instruction issued by t between c' and c. Non-transactional instructions that appear outside transactions behave like single-instruction transactions that take immediate effect (aborting all transactions that write-read or write-write conflict on a cache line basis). Concretely, this corresponds to adding the transitions in Fig.7 to Δ, and to replacing $c = tld(v)$ (resp. $c = tst(v)$) by $c \in \{ld(v), tld(v)\}$ (resp. $c \in \{st(v), tst(v)\}$) in case 1 (resp. case 2) of Fig.4. With these modifications, we capture the effect of any non-transactional instruction issued by the programs (the instruction c) using (sequences of) transactional statements (the effect d).

III. CORRECTNESS PROPERTIES OF THE GENERATED SEQUENCES

We formally define the safety and liveness we aim to verify.

First we introduce some notions we use in the definitions of the correctness properties. Two statements s_1 (from transaction x) and s_2 (from transaction y) are said to *conflict* if either: s_1 is a global read of a variable v (which means v has not been written in x before), y writes to v and s_2 is a commit, or if both x and y write to v and both s_1 and s_2 are commits. Two words w and w' with the same, possibly reordered, statements are *conflict equivalent* if for every pair s, s' of statements in w, if s conflicts with s' and s occurs before s', then s also occurs before s' in w'. Two conflict equivalent words w and w' are *strictly equivalent* if for every two transactions x, y in w, if $x <_w y$ then $x <_{w'} y$. A word w is *strictly serializable* (resp. *abort consistent*) if there exists a sequential word w' such that $com(w)$ (resp. w) is *strictly equivalent* to w'. We say

that FLEXTM is strictly serializable (resp. abort consistent) if all generated words (i.e. all $w \in \cup_{n,k} \cup_{p \in P_k^n} \mathcal{L}(p)$) are strictly serializable (resp. abort consistent). Observe that abort consistency implies strict serializability.

Obstruction [18] and *livelock freedom* [17] are liveness properties and capture requirements on the progress of transactions. Intuitively, obstruction freedom requires that a transaction is guaranteed to make progress if all other transactions are suspended. Formally, $\wedge_{t \in T}(\Box\Diamond((com, t) \vee (c, u)) \vee \Diamond\Box\neg(abort, t))$ with $c \in \hat{C}$ and $u \neq t$. Livelock freedom requires that there is always progress in any infinite trace. Formally, $\Box\Diamond(\vee_{t \in T}(com, t)) \vee \Diamond\Box(\wedge_{t \in T}\neg(abort, t))$.

Given T and V, observe from the description of FLEXTM in Section II, that the only way to generate a word that violates obstruction freedom is to go through a loop that has an abort but no commit and where all statements are from the same thread. Showing the absence of such a loop, for any sets T and V therefore proves obstruction freedom. Similarly, showing the absence, for any sets T and V, of a loop that has no commit but where all threads have at least an abort statement, proves livelock freedom.

IV. SMALL MODEL THEOREM AND REQUIRED PROPERTIES

We recall and state the properties introduced in [22] in order to justify a number of small model theorems. In short, these theorems ensure that if the correctness properties introduced in Section III hold for all programs running on FLEXTM with two threads and two variables, then they will hold for any program with any number of threads or variables. For lack of space, we do not prove the properties in this Section.

A. Safety properties

For strict serializability, it is enough to show that FLEXTM satisfies the properties P1, P2, P3 and P4 below in order to apply the small model theorem. Let $w \in \mathcal{L}(p)$ for some program $p \in P_k^n$. Let $\rho = (s_0, l_0) \xrightarrow{\delta_0} (s_1, l_1) \xrightarrow{\delta_1} \ldots (s_m, l_m)$ be generated by some program $p \in P_k^n$ with $w = \rho_{|\hat{S}}$.

P1. Assume w has no aborting transactions. If for all transactions $x \in Tx_t(w)$ and $y \in Tx_u(w)$ we have that either $x <_w y$, or that $y <_w x$, then the word $w' = w[t/u]$ obtained by renaming all transactions of thread u to be from thread t is in $\mathcal{L}(p')$ for some program $p' \in P_k^{n-1}$.
lemma 1: FLEXTM satisfies P1.

P2. Let X be a subset of the set of committing and pending transactions in w. The transaction projection of w on X is in $\mathcal{L}(p')$ for some program p'.
lemma 2: FLEXTM satisfies P2.

P3. Let w be a generated word that have no aborting transaction. For a program p, the variable projection of p on $V' \subseteq V$ is the program obtained by removing, from all unrolled threads in p, all reads and writes to the variables which are not in V'. The variable projection of p on V' is generated by p' where p' is the projection of p on V'.
lemma 3: FLEXTM satisfies P3.

P4. If a word w generated by some program p is an extension of some word w_1, i.e., $w = w_1 \cdot s$, where w_1 is strictly serializable and s is a non aborting statement of the only pending transaction of w_1, then there is a word w_2 such that w_1 is strictly equivalent to w_2 and $com(w_2)$ is sequential and $w_2 \cdot s$ is also generated by p.
lemma 4: FLEXTM satisfies P4.

In a similar manner, if the properties P5 and P6 hold, then it is enough to show obstruction freedom for all programs on two threads and two variables to deduce that FLEXTM is obstruction free.

Let $w = w_1 \cdot w_2$ be an infinite word in $\mathcal{L}(p)$ for some program p. Assume there are no pending transactions from w_1 that have a statement in w_2, and all statements in w_2 belong to the same thread.

P5. Let w_1' be the projection of w_1 on non aborting transactions. Then $w_1' \cdot w_2 \in \mathcal{L}(p')$ for some program p'. Moreover, if w_1 has no aborting transactions, there exists a word $w' = w_1'' \cdot w_2$ in $\mathcal{L}(p)$ where w_1'' is obtained by projecting w_1 to transactions of thread t, where t has instructions in w_1.
lemma 5: FLEXTM satisfies P5.

P6. There exists a word $w' = w_1 \cdot w_2'$ such that w_2' is obtained by projecting on variable $\{v\}$, where v is accessed in w_2, and w' is in $\mathcal{L}(p)$ for some program p. Moreover, if w_1 has no aborting transactions, then the word $w' = w_1' \cdot w_2$ is in $\mathcal{L}(p')$ for some program p', where w_1' is obtained by projecting on the variables of w_1 accessed in w_2.
lemma 6: FLEXTM satisfies P6.

V. EXPERIMENTS

We build four models to capture the behavior of FLEXTM and two reference models capturing most general descriptions for both strict serializability and for abort consistency [22]. Every model involves two threads and two variables and is represented by an automaton. For the FLEXTM models, we implement both lazy and eager conflict detection modes and take non-transactional instructions into account. For each mode we build two models: one has two cache lines with one variable in each line, while the other has only one cache line with both variables in the line. We handle non-transactional instructions and invisible reads as described in II-E and II-B.

We use *VATA Library* [24], a tool that implements the antichain based algorithm for checking language inclusion of (tree and word) automata [23]. The results are obtained on a dual core laptop PC with 2GB of RAM. As described in TABLE I, we establish the inclusion of the languages of the four FLEXTM automata in the languages of each of the two reference automata. This ensures that all the traces generated by the models of the two modeled varieties FLEXTM (i.e., eager and lazy) are both strictly serializable and abort consistent. Using the small model theorem introduced in Section IV, this establishes that the two modeled varieties of conflict detection and management in FLEXTM satisfy both strict serializability and abort consistency. We also modified the models in order to allow write-write conflicts. This is a commonly allowed conflict in software transactional

TABLE I
NUMBER OF STATES AND SECONDS REQUIRED TO ESTABLISH THE INCLUSION OF THE LANGUAGES OF THE AUTOMATA IN THE FIRST COLUMN IN THE LANGUAGES OF THE AUTOMATA IN THE FIRST ROW.

$\subseteq$	strict serializability (24173 states)	abort consistency (53084 states)
Eager-1-cache (695 states)	20s	39s
Eager-2-cache (19985 states)	29s	50s
Lazy-1-cache (1708 states)	20s	38s
Lazy-2-cache (57755 states)	41s	65s

memories. Our implementation automatically exhibits the following non serializable execution $(tst(v_1), t_1) \cdot \underline{(tld(v_2), t_1)} \cdot ((tst(v_2), t_2)) \cdot \underline{(tld(v_1), t_2)} \cdot (com, t_1) \cdot (com, t_2)$ where the underlined statements are accesses due to invisible reads.

We verify liveness properties by looking for loops in the generated models. We establish that FLEXTM is obstruction free by checking the absence of loops with an $abort$, no com and where all statements are from the same thread (see Section III). In a similar manner, we refute livelock-freedom by finding loops that have no com but where all threads have at least an $abort$ statement; for instance (discarding invisible reads for clarity) $(tld(v), t_1) \cdot ((tst(v), t_2) \cdot (abort, t_1) \cdot (tst(v), t_1) \cdot (abort, t_2))^\omega$ for the eager mode, and $(tst(v), t_1) \cdot (tst(v), t_2) \cdot ((check, t_1) \cdot (inval, t_1) \cdot (abort, t_2) \cdot (tst(v), t_2) \cdot (check, t_2) \cdot (inval, t_2) \cdot (abort, t_1) \cdot (tst(v), t_1))^\omega$ for the lazy mode with non-atomic commit.

VI. CONCLUSION

We have proposed faithful and detailed models for both the eager and the lazy modes of the FLEXTM hybrid transactional memory. We have considered intricate aspects that are specific to hardware and hybrid transactional memories such as invisible reads and cache line based conflict detection and tracking. Then, we have adapted a small model based approach developed for software transactional memories in order to establish a number of properties allowing simplification of safety and liveness verification for any numbers of concurrent transactions of arbitrary lengths and accessing arbitrary numbers of variables. This has resulted in representations on which it was crucial to apply state-of-the-art techniques for checking inclusion of regular languages in order to automatically establish, for the first time, strict serializability, abort consistency and obstruction freedom for a hybrid transactional memory. This work can be extended by establishing a high level language for describing several hardware and hybrid transactional memories. We believe a challenging task is to automatically establish the properties allowing the application of small model approaches, and to extract the corresponding systems and carry out the actual verification.

ACKNOWLEDGMENT

Work supported by the Uppsala Programming for Multicore Architectures Research Center (UPMARC) and by the CENIIT Software Model Checking in the Multicore Era project.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993, pp. 289–300.
[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *HPCA*. IEEE Computer Society, 2005, pp. 316–327.
[3] L. Hammond, V. Wong, M. K. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *ISCA*. IEEE Computer Society, 2004, pp. 102–113.
[4] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: log-based transactional memory," in *HPCA*. IEEE Computer Society, 2006, pp. 254–265.
[5] R. Rajwar, M. Herlihy, and K. K. Lai, "Virtualizing transactional memory," in *ISCA*. IEEE Computer Society, 2005, pp. 494–505.
[6] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *DISC*, ser. LNCS, vol. 4167. Springer, 2006, pp. 194–208.
[7] K. Fraser and T. L. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, 2007.
[8] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III, "Software transactional memory for dynamic-sized data structures," in *PODC*. ACM, 2003, pp. 92–101.
[9] V. J. Marathe, W. N. S. III, and M. L. Scott, "Adaptive software transactional memory," in *DISC*, ser. LNCS, vol. 3724. Springer, 2005, pp. 354–368.
[10] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime," in *PPOPP*. ACM, 2006, pp. 187–197.
[11] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *ASPLOS*. ACM, 2006, pp. 336–346.
[12] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. D. Nguyen, "Hybrid transactional memory," in *PPOPP*. ACM, 2006, pp. 209–220.
[13] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. G. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An effective hybrid transactional memory system with strong isolation guarantees," in *ISCA*. ACM, 2007, pp. 69–80.
[14] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott, "An integrated hardware-software approach to flexible transactional memory," in *ISCA*. ACM, 2007, pp. 104–115.
[15] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *ISCA*. IEEE, 2008, pp. 139–150.
[16] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd edition*. Morgan & Claypool, 2010.
[17] J. H. Anderson, Y.-J. Kim, and T. Herman, "Shared-memory mutual exclusion: major research trends since 1986," *Distributed Computing*, vol. 16, no. 2-3, pp. 75–110, 2003.
[18] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *ICDCS*. IEEE Computer Society, 2003, pp. 522–529.
[19] G. Delzanno, "Automatic verification of parameterized cache coherence protocols," in *CAV*, ser. LNCS, vol. 1855. Springer, 2000, pp. 53–68.
[20] E. A. Emerson and V. Kahlon, "Exact and efficient verification of parameterized cache coherence protocols," in *CHARME*, ser. LNCS, vol. 2860. Springer, 2003, pp. 247–262.
[21] P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine, "Regular model checking without transducers (on efficient verification of parameterized systems)," in *TACAS*, ser. LNCS, vol. 4424. Springer, 2007, pp. 721–736.
[22] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh, "Model checking transactional memories," in *PLDI*. ACM, 2008, pp. 372–382.
[23] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar, "When simulation meets antichains," in *TACAS*, ser. LNCS, vol. 6015. Springer, 2010, pp. 158–174.
[24] O. Lengál, J. Simácek, and T. Vojnar, "Vata: A library for efficient manipulation of non-deterministic tree automata," in *TACAS*, ser. LNCS, vol. 7214. Springer, 2012, pp. 79–94.
[25] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *ISCA*. ACM, 1984, pp. 348–354.