We propose two fine grained models for the FlexTM memory. More concretely, we propose parameterized models atomicity illusion. [6]–[10], or hybrid [11]–[14]) infrastructure that maintains the underlying Transactional Memory (hardware [1]–[5], software if needed. The concurrent transactions run on top of an Otherwise, the transaction is aborted and may be restarted successful transaction takes effect and is said to be committed. (i.e., a transaction) boils down to a finite sequence of loads the blocks that are to be executed atomically. Each such block the offered parallelism while giving the programmer the illu-

the Transactional Memory (TM) approach takes advantage of eases these 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 verifica-

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
We build these models and obtain automata with tens of thousands of states. We establish obstruction freedom and refute livelock responses to invisible reads and false sharing. The aspects are automatically captured 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 lazy 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 multi-processor 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 $C \cup \{abott\}$ and $\hat{C} \times \hat{T}$ respectively. A binary tree $\theta : \{0,1\}^* \rightarrow C$ 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 $\Theta$ 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 $T$. $P^n_k$ 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 $S(v)$ to mean the cache line associated with $v$, and $S(V)$ to mean the set of all cache lines. We also write $v' \in S(v)$ 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 $S(v)$ to its cache. This is known as invisible reads and amounts to simultaneously reading all variables associated to $S(v)$. In the descriptions that follow, and given a thread $t$, we use $(v',t \in S(v))(tl(d(v'),t))$ to mean a sequence

| thread $t_1$ | $tl(d(a))$ | $r_0/t_1$ | $r_1/t_1$ | $r_0$ | $r_1$ | $r_0/r_1$ | $r_1/r_0$ |
| thread $t_2$ | $tl(d(a))$ | $r_0/t_1$ | $r_1/t_1$ | $r_0$ | $r_1$ | $r_0/r_1$ | $r_1/r_0$ |

**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.

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.
(tl(d(v₁), t) · (tl(d(v₂), t) · · · (tl(d(vₙ), t) of instructions where thread t reads, in some fixed arbitrary order, all variables in $\$v$). For example, we write $(v_i \in \$v)(tl(d(v_i), t) · (ts(t), 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 $(Q, \Delta, s_{init})$. A state s in Q is a tuple $(status, sigs, cst, cst_v)$ where:

- $status : T \rightarrow \{idle, active, check, abort\}$ keeps track of the status of the current transaction of each thread in T.
- $sigs$ is a pair $(r_{sig}, w_{sig})$ of mappings $T \rightarrow 2^{\$v}$ 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 $(ru, wr, uw)$ of three mappings $T \rightarrow 2^T$ 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.
- $cst_v$ is a pair of mappings used during the nonatomic commit phase, they hold copies of $uw$ and $wv$.

![Fig. 2. Possible status changes for a thread t in the lazy mode of FLEXTM.](image)

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 $\Delta$ is a tuple $(s, c, d, r, s')$, where $s, s' \in Q$, $c \in C$ and $d \in \{(c_i, t) \cdots (c_j, t)\mid \{|v| : 1 \leq i \leq m : c_i \in C \text{ and } t \in T\} \cup \{(abort, check, invalid) \times T\}$ with $r \in \{0, 1, \bot\}$. 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

![Fig. 3. Thread t with idle status issues an instruction c](image)

1) $c = tl(d(v)) and \ U_{cst} = \{u \neq t \text{ and } v(\in \ w_{sig}(u))\}$, then:
   a) $\ status' = status[t \leftarrow \ active]$;
   b) $r'_{sig} = r_{sig}[t \leftarrow (r'_{sig}(t) \cup \$v)]$;
   c) $\ ru' = ru[t \leftarrow (ru(t) \cup U_{cst})]$;
   d) $\ uw' = uw[t \leftarrow (uw(t) \cup \ U_{cst})]$, and
   e) $\ d = (v_i \in \$v)(tl(d(v_i), t) · (ts(t), t)) and r = 1);
2) $c = ts(t)$ with $\ U_{cst} = \{u \neq t \text{ and } v(\in \ r_{sig}(u))\}$ and $\ U_{cst} = \{u \neq t \text{ and } \$v(\in \ w_{sig}(u))\}$, then:
   a) $\ status' = status[t \leftarrow \ active]$;
   b) $\ u'_{sig} = w_{sig} \times (u(\notin \ U_{cst}) \cup \$v))$;
   c) $\ ru' = ru[t \leftarrow (ru(t) \cup \ U_{cst})]$;
   d) $\ uw' = uw[t \leftarrow (uw(t) \cup \ U_{cst})]$, and
   e) $\ d = (v_i \in \$v)(tl(d(v_i), t) · (ts(t), t)) and r = 1);
3) $c = corn$ then $d = (c, t)$ and $r = 1$;

![Fig. 4. Thread t with active status issues an instruction c](image)

For example, Fig.5 describes a thread t running the nonatomic commit procedure. In this description, t aborts conflicting transactions 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 uw and uw, 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
If there is a thread \( t \neq t' \) with \( w \in (u_{wr}(t) \cup u_{rw}(t)) \), then:

1. \( u_{wr}(t) = u_{rw}(t) \cup \{ w \} \)
2. \( u_{rw}(t) = u_{wr}(t) \cup \{ w \} \)
3. \( \text{status}(u) \in \{ \text{active, check} \} \)

   - \( \text{status}'(u) = \text{status}(u) \cup \{ \text{abort} \} \)
   - \( d = (\text{intra}(u), t) \) and \( r = \Delta \)

If \( (u_{wr}(t) \cup u_{rw}(t)) = \emptyset \) but \( (u_{wr}(t) \cup u_{rw}(t)) \neq \emptyset \), then:

1. \( u_{wr}(t) = u_{rw}(t) \cup \{ w \} \)
2. \( u_{rw}(t) = u_{wr}(t) \cup \{ w \} \)
3. \( \text{status}'(u) = \text{status}(u) \cup \{ \text{abort} \} \)
   - \( d = (\text{chck}(e), t) \) and \( r = \Delta \)

If \( (u_{wr}(t) \cup u_{rw}(t) \cup u_{rw}(t) \cup u_{wr}(t)) = \emptyset \), then:

1. \( \text{status}'(u) = \text{status}(u) \cup \{ \text{idle} \} \)
2. \( d = (\text{com}(t), t) \) and \( r = 1 \)

Fig. 5. Thread \( t \) runs the non-atomic commit, i.e., \( \text{status}(t) = \text{check} \).

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 \( P_k \) with a scheduler \( \sigma : N \rightarrow [0, 1 \leq i \leq n] \) on a FlexTM is defined as follows. A run is a sequence \( \rho = (s_0, l_0) \xrightarrow{\delta} (s_1, l_1) \xrightarrow{\delta_1} \ldots \) where \( s_i \in Q \), \( l_i = (l_i[1], \ldots, l_i[n]) \in ([0,1]^*)^n \), with \( s_0 = s_{tinit} \) and \( l_0 = (e, \ldots, e) \), and such that for all \( g \geq 0 \), \( \delta_j = (s_j, c_j, d_j, r_j, s_{j+1}) \) in \( \Delta \) with: \( t_{\sigma(j)} = \text{thread}O(d_j), \)

\( c_j = \theta([\sigma(j)], l_j[\sigma(j)]) \) and \( r_l \in T, if t_{\sigma(j)} \text{ or } r_j = 1 \)

then \( l_{j+1}[\sigma(j)] = l_j[\sigma(j)] \), otherwise \( l_{j+1}[\sigma(j)] = l_j[\sigma(j)] \).

We write \( \rho_{1b} \) to mean the projection of the sequence \( d_0, d_1 \ldots \) on \( S^* \). A word \( w \in S^* \) is said to be generated by a program \( p \), and write \( w \in L(p) \), if \( w = \rho_{1b} \) for some run \( \rho \) of \( p \) with respect to some scheduler.

D. Transactions

Given a finite word \( w \in S^* \), we project on a thread \( t \) and obtain a sequence \( w_{t} \). We define terminating statements to be the statements in \( \{ \text{com}(t), \text{abort}(t) \} \) 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 \( w \in L(p) \) 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 \( T_{tx}(w) \) 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 \( x \in T_{tx}(w) \), \( x \) is said to be committing if \( (\text{com}(t), t) \in x \), i.e. if the statement \( (\text{com}(t), t) \) appears in \( x \), aborting if \( (\text{abort}(t), t) \in T_{tx}(w) \), and pending otherwise. Given a word \( w \), we write \( \text{com}(w) \) 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

Fig. 7. Thread \( t \) with \( \text{idle} \) status issues a non-transactional instruction \( c \) 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 \( V \) is the largest 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 \( x \prec w y \) to mean that the initiating statement of \( y \) appears in \( w \), after the terminating statement of \( x \). A word is \( \text{sequential} \) if for every pair of transactions \( x, y \) in the word, either \( x \prec w y \) or \( y \prec w x \).

E. Non-transactional instructions

In reality, FlexTM also allows non-transactional instructions in programs. A non-transactional instruction is either a read \( \text{ld}(v) \) or a write \( \text{st}(v) \) 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 transactional 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 \( \Delta \), and to replacing \( c = \text{ld}(v) \) (resp. \( c = \text{st}(v) \)) by \( c \in \{ \text{ld}(v), \text{ld}(v) \} \) (resp. \( c \in \{ \text{st}(v), \text{st}(v) \} \)) 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 \( d \)).
that FLEXTM is strictly serializable (resp. abort consistent) if all generated words (i.e., all \( w \in \bigcup_{n,k} \bigcup_{p \in P^n_k} \mathcal{L}(p) \)) 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, \( \forall t \in T(\Box (\Diamond (\text{commit}, t) \lor (c, u)) \lor \Box \neg (\text{abort}, t)) \) with \( c \in C \) and \( u \neq t \). Livelock freedom requires that there is always progress in any infinite trace. Formally, \( \Box (\Diamond (\text{commit}, t) \lor \Box (\Diamond (\text{abort}, t)) \).

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 \( p \in P^n_k \). Let \( \rho = (s_0, l_0) \overset{\delta_0}{\rightarrow} (s_1, l_1) \overset{\delta_1}{\rightarrow} \ldots \overset{\delta_{m-1}}{\rightarrow} (s_m, l_m) \) be generated by some program \( p \in P^n_k \) with \( w = \rho_{[0, l]} \).

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

**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 \( \text{commit}(w_2) \) 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 \( w'_1 = w'_1 \cdot w_2 \in \mathcal{L}(p) \) 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 \( w'_2 \) 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
memories. Our implementation automatically exhibits the following non serializable execution \((tst(v_1), t_1) \cdot (tld(v_2), t_1) \cdot ((tst(v_2), t_2)) \cdot (tld(v_1), t_2) \cdot (com, t_1) \cdot (com, t_2)\) 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 \textit{abort}, \textit{no com} and where all statements are from the same thread (see Section III). In a similar manner, we refute livelock-freeness by finding loops that have no \textit{com} but where all threads have at least an \textit{abort} statement; for instance (discarding invisible reads for clarity) \((tld(v), t_1) \cdot (tst(v), t_2) \cdot (abort, t_1) \cdot (tst(v), t_1) \cdot (abort, t_2)^2\) for the eager mode, and \((tst(v), t_1) \cdot (tst(v), t_2) \cdot ((check, t_1) \cdot ( inval, t_1) \cdot ( abort, t_2) \cdot (tst(v), t_2) \cdot (check, t_2) \cdot ( inval, t_2) \cdot (abort, t_1) \cdot (tst(v), t_1))^2\) 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