Parameterized Verification of Transactional Memories *

Michael Emmi Rupak Majumdar Roman Manevich
University of California, Los Angeles
{mje,rupak,rumster}@cs.ucla.edu

Abstract

We describe an automatic verification method to check whether transactional memories ensure strict serializability—a key property assumed of the transactional interface. Our main contribution is a technique for effectively verifying parameterized systems. The technique merges ideas from parameterized hardware and protocol verification—verification by invisible invariants and symmetry reduction—with ideas from software verification—template-based invariant generation and satisfiability checking for quantified formulæ (modulo theories). The combination enables us to precisely model and analyze unbounded systems while taming state explosion.

Our technique enables automated proofs that two-phase locking (TPL), dynamic software transactional memory (DSTM), and transactional locking II (TL2) systems ensure strict serializability. The verification is challenging since the systems are unbounded in several dimensions: the number and length of concurrently executing transactions, and the size of the shared memory they access, have no finite limit. In contrast, state-of-the-art software model checking tools such as BLAST and TVLA are unable to validate either system, due to inherent expressiveness limitations or state explosion.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent Programming; D.2.4 [*Software engineering*]: Software/Program Verification

General Terms Reliability, Verification

Keywords Transactional memory, Parameterized verification

1. Introduction

Transactional Memories Transactional memory (TM) [19, 23] is a model of concurrent programming which allows programmers to write code with coarse-grained atomic blocks whose interactions, and possible conflicts, are managed at run time by a transaction manager. The transaction manager provides the illusion that application-level transactions (atomic sequences of data reads/writes, followed by a commit) execute sequentially, at the expense of tracking, and potentially aborting or re-executing, conflicting transactions.

By freeing the programmer from low-level book-keeping for concurrency, TM systems hold the promise of higher programmer productivity without sacrificing highly concurrent operations. Since TM systems form the core algorithms on which the entire software stack depends, it is crucial that they are implemented correctly. This is not trivial, as transactional memories employ sophisticated mechanisms to ensure efficiency, and in fact, several bugs have been uncovered in existing TM implementations [12].

Ensuring Strict Serializability The above considerations indicate that the *formal verification* of TM systems should be a rich and important target for software verification research. Indeed, there have been several attempts to verify TM algorithms [6, 15]. What is missing, however, from the initial research is the *automatic* verification of transactional memories against a correctness property such as *strict serializability* [27]. In principle, by encoding the TM algorithm and the strict serializability specification as transition systems, a state-exploration technique such as model checking can prove the property automatically. Unfortunately, TM algorithms are unbounded in several dimensions: in the number of interacting threads and shared memory locations, and in the number and length of transactions. This precludes simple applications of model checkers, and two approaches have been taken in previous work. First, Cohen et al. [6, 7] and Taşiran [33] manually specify the refinement relation between implementation and specification in order to use an automated theorem prover. This annotation requires a thorough understanding of both the implementation and specification, and like most manual verification effort, carries a high cost. Second Guerraoui et al. [15, 16] rely on a meta-theorem on the structure of the TM algorithm to prove a "small model" result that says that the TM algorithm is correct if and only if it is correct for two threads and two locations. The small instance can be discharged using finite-state model checking. While this goes a long way, the missing piece is the (essentially manual) proof of the meta-theorem for each new TM implementation.

Parameterized Verification In this paper, we describe a novel technique for the *fully automatic, parameterized* verification of TM implementations. Our starting point is the formalization of TM systems [15] and the deterministic specification of strict serializability [16]. Given a TM implementation parameterized by the number of threads n and the number of shared locations k, our goal is to automatically construct a family of simulation relations that demonstrates, *for all* $n > 0$ and $k > 0$, that the TM implementation with n threads and k locations refines the strict serializability specification. Using standard techniques, this reduces to a uniform invariant verification problem on the composition of the TM implementation and the strict serializability specification.

Parameterized verification of program invariants is undecidable in general [1], and our models do not fall into a known syntactic class with a finite-model property, so we adopt a sound but potentially incomplete technique. The standard approach [25] of showing that a property ψ is a *uniform* invariant of a family M of parameterized systems (i.e., ψ is an invariant for every $M(n, k)$ in the family) finds a *uniform inductive invariant* φ of M that implies ψ, thus breaking

* This research was sponsored in part by the NSF grants CCF-0546170 and CCF-0702743, and DARPA grant HR0011-09-1-0037.

PLDI '10 June 5–10 2010, Toronto, Ontario, Canada.

verification into two sub-problems: (A) to come up with potential invariants φ, and (B) to check that φ is a uniform inductive invariant of M and implies ψ. These problems are hard, since parameterized systems typically involve quantified transitions and invariants.

We propose solutions to problems (A) and (B) that exploit the structure of transactional memory implementations in the following way. In transactional memory systems, *conflicts* (e.g., distinct threads reading from or writing to the same location) are a central notion. We notice that (i) the transition relation of these systems is *symmetric* (i.e., "thread identifiers" are not made explicit), and (ii) in the transition relations of these systems, an executing thread, e.g., t, interacts only with either arbitrary threads u, or threads that are *conflict-adjacent* (i.e., threads u that conflict with t, but not threads u' that conflict with u but not t). The symmetry suggests that we can reason about *arbitrary* threads (and memory locations), while conflict adjacency suggests we can restrict reasoning to small *neighborhoods* surrounding particular threads. We propose the following techniques that utilize these observations.

A. Candidate Invariant Generation Inspired by the method of *verification by invisible invariants* [2, 28], we generate candidate invariants from the set of reachable states of *bounded instantiations* $M(n_0, k_0)$ of the parameterized system M for small fixed values of n_0 and k_0. On its own this is not enough, since the resulting candidate invariants generated directly from the reachable states are too large to be effectively discharged in the following validation step, and as suggested by Arons et al. [2], are often too specific to be uniform invariants—the arbitrary Boolean structure of the formula prevents separating out "parts" of the candidate that are uniform invariants from parts that are specific to the particular instance.

We fix these problems by combining the technique with a novel application of *template-based invariant generation* [8, 32]. In particular, we fix a *template language* of formulæ, and look for invariants of $M(n_0, k_0)$ that are expressible as a *conjunction* of templates instantiated with system predicates. The use of conjunctive templates has two advantages. First, even if a conjunction of candidate invariants is not a uniform invariant, we can generate weaker candidates by discarding conjuncts which are not uniform invariants. (Indeed this is necessary, since in our experiments, often fewer than half of the small candidate invariants could be validated.) Second, we can *bias* the search toward smaller invariants. Intuitively, the small neighborhood observation of transactional memory systems suggests that interesting system behaviors are exercised with few threads and memory locations, and that small template formulæ (e.g., involving few quantified variables) obtained by generalizing the reachable states of small instances should describe the interesting behaviors. Symmetry suggests that the resulting invariants apply to any arbitrary threads and locations.

B. Invariant Validation Checking that candidate invariants are uniform reduces to checking the validity of quantified first order formulæ. While this validity checking problem is undecidable in general, relying on well-engineered automatic (sound, but incomplete) theorem provers [9, 13, 34] has worked for our experiments.

We have implemented the steps above in a verification tool built on top of TVLA's abstract reachability engine [24] and the SPASS theorem prover [34], and we have used our implementation to automatically verify the correctness of several TM systems: two-phase locking, DSTM [20], and TL2 [10]. The end-to-end verification for TL2, the most complicated algorithm, took about 100 minutes, finding a quantified invariant with 416 conjuncts. The size of the invariant shows the utility of having an automatic verification algorithm. In contrast, our attempts to verify the same algorithms with state-of-the-art software model checkers such as BLAST and TVLA, as well as our implementation of the algorithm by Arons et al. [2], failed due to expressiveness limitations (e.g., BLAST cannot infer or reason with quantified invariants) or due to scalability reasons (e.g., with manually provided instrumentation predicates, TVLA timed out on the examples).

The key contributions of this paper are summarized as:

1. We develop an algorithm capable of automatically verifying that common TM systems ensure strict serializability, and
2. We introduce an effective invariant generation technique by combining verification by invisible invariants with template-based invariant generation, effectively extending the applicability of existing parameterized verification techniques.

Although we have applied our technique to transactional memory systems, our algorithmic contributions are general, and are likely to be useful in many other instances of parameterized software verification.

2. Transactional Memory Verification

Transactions Let Thd be a set of *threads* and Loc a set of shared memory *locations*. For each thread $t \in Thd$, let $A_t = \{read(t, v), write(t, v) : v \in Loc\} \cup \{commit(t), abort(t)\}$ be the set of system *actions performed by* t, and $A = \bigcup\{A_t : t \in Thd\}$. A system *trace* is a finite sequence $w \in A^*$.

Let $w \in A^*$ be a system trace. The *projection* $w|_t = a_1 a_2 \ldots \in A_t^*$ of w on a thread $t \in Thd$ is the subsequence of w consisting only of actions performed by t. An action a_i is *finishing in* $w|_t$ if it is a $commit$ or $abort$, and is *starting in* $w|_t$ if $i = 1$, or a_{i-1} is finishing in $w|_t$.

A consecutive subsequence $u = b_1 b_2 \ldots b_m$ of $w|_t$ is a *transaction* of t if (i) b_1 is starting in $w|_t$, (ii) if b_i is finishing in $w|_t$ then $i = m$, and (iii) if b_m is not finishing in $w|_t$ then b_m is the last action of $w|_t$. The transaction u is *committing* when b_m is a $commit$, *aborting* when b_m is an $abort$, and *unfinished* otherwise. For $w \in A^*$, we define the word $com(w) \in A^*$ to be the subsequence of w consisting of every action of committing transactions of w. If $b_i = read(t, v)$ (resp., $b_i = write(t, v)$) for some i we say u *reads* (resp., *writes*) to location v, and call b_i a *read* (resp., *write*) to v. If b_i is a read to v and for some $j < i$, b_j is a write to v, we call b_i a *local* read; b_i is otherwise a *global* read. By this definition of transaction, we restrict our attention to the case where all reads and writes to locations $v \in Loc$ occur within some transaction.

Strict Serializability We consider a correctness criterion for transactional systems called *strict serializability* [15, 16, 27] (SS). Intuitively, SS insists that each system trace may be reordered (while preserving a *conflict order*, defined below) into an observationally equivalent sequence such that transactions of distinct threads do not overlap and the order of non-overlapping transactions is preserved. This notion is a useful abstraction for programmers of concurrent systems since each transaction can then be thought to execute in isolation. We formalize SS following Guerraoui et al. [15].

Let $w \in A^*$ and let u_1 and u_2 be two transactions in w, possibly of different threads. We say u_1 *precedes* u_2 in w, written $u_1 <_w u_2$, when the last action of u_1 occurs before the first action of u_2 in w. A trace in which transactions are totally ordered by $<_w$ is called *sequential*.

Actions a_1 of a transaction u_1 and a_2 of a transaction u_2 with $u_2 \neq u_1$ are said to *conflict* in a trace w when there exists a location $v \in Loc$ such that either (i) a_1 is a global read of v, u_2 writes to v, and a_2 is a $commit$ action, or (ii) a_1 and a_2 are $commit$ actions, and both u_1 and u_2 write to v. This notion of conflict assumes the *deferred update semantics* of transactions [23], where values written by a transaction u are not observed by other transactions until u commits.

Two traces $w_1 = a_1 a_2 \ldots$ and w_2 are *strictly equivalent* when

(i) $w_1|_t = w_2|_t$ for every thread $t \in Thd$,

(ii) a_i occurs before a_j in w_2 whenever $i < j$ and a_i and a_j conflict in w_1, and

(iii) if $u_1 <_{w_1} u_2$ and u_1 is not unfinished then $u_2 \not<_{w_2} u_1$.

A trace w is said to be *strictly serializable* when there exists a sequential trace w' that is strictly equivalent to $com(w)$. The set of strictly serializable traces is written L_{SS}.

Transactional Memory A *multi-threaded program* $\Pi_{n,k}$ with n threads and k locations is an n-tuple $\langle \pi_1, \ldots, \pi_n \rangle$ of programs, where for $i = 1, \ldots, n$ the program $\pi_i : \mathbb{B}^* \to A_{t_i}$ is a map from infinite binary trees to actions performed by thread t_i. The representation of programs as infinite binary trees, where intuitively the left branch denotes "successful" execution and the right branch denotes "aborted" execution of an operation, abstracts from specific control flow structures. Let $\Pi = \bigcup\{\Pi_{n,k} : n, k \in \mathbb{N}^{>0}\}$ be the set of all multi-threaded programs.

A *transactional memory (TM)* M is a function from multi-threaded programs in $\pi \in \Pi$ to sets $M(\pi) \subseteq A^*$ of traces. A TM M is said to *ensure strict serializability* if for each multi-threaded program $\pi \in \Pi$, and each trace $w \in M(\pi)$, w is strictly serializable, i.e., $M(\pi) \subseteq L_{SS}$. The verification problem we study here is:

Problem 1 (TM Verification)**.** *Does a given transactional memory M ensure strict serializability?*

Typical software transactional memory systems ensure strict serializability [15]. For example, a two-phase locking (2PL) protocol ensures that its traces are serializable by prohibiting overlapping transactions from accessing write-open (resp., read- or write-open) memory locations for reading (resp., writing). More complex systems (i.e., those allowing more concurrent behavior) such as Dynamic Software Transactional Memory [20] (DSTM), and Transactional Locking II [10] (TL2), use more intricate mechanisms to ensure strict serializability for which it is not obvious whether strict serializability is ensured.

3. Parameterized Systems

The TM verification problem is an instance of *parameterized verification*, as the property must be proved for programs with *arbitrarily* many threads and *arbitrarily* many locations. We now describe our verification approach on a logical representation for parameterized systems.

Systems We fix a set $\mathbb{T} = \{Thd, Loc\}$ of sorts of *threads* and *locations*, and a sorted set $X = \{t, t_1, t_2, \ldots\} \cup \{v, v_1, v_2, \ldots\}$ of logical variables, where by convention $t, t_1, \ldots$ have sort Thd and $v, v_1, \ldots$ have sort Loc. When not clear from the context, we explicitly write, e.g., $t : Thd$ or $v : Loc$. We assume the only logical variables occurring in formulæ come from X.

For a set of ranked and sorted predicates P, a *P-formula* (resp., *P-sentence*) is a first-order logical formula (resp., sentence) over a language consisting of the equality symbol "$=$" and the predicates in P. We additionally define the set $P' = \{p' : p \in P\}$ of ranked and sorted predicates p renamed to p'. For a P-formula φ we write φ' for the P'-formula $\varphi' = \varphi[P'/P]$, where each predicate p of φ is renamed to p'. We omit the ranks and sorts of predicates for readability. We write $\varphi \approx \psi$ when φ and ψ are syntactically equal formulæ.

For $n, k \in \mathbb{N}^{>0}$, the *parameterized system* $M(n, k)$ of size $\langle n, k \rangle$ is a tuple $\langle U, P, A, \Theta, g, \rho \rangle$ consisting of:

- a sorted universe U of n threads and k memory locations,
- a finite set P of ranked and sorted predicates with sorts in $\mathbb{T}$,
- a finite set A of actions $a(\vec{x})$ where $\vec{x}$ is a vector of sorted logical variables in X,
- a P-sentence Θ specifying the initial condition,
- for each action $a(\vec{x}) \in A$, a P-formula $g_a(\vec{x})$ (called the *guard* of a), and
- for each action $a(\vec{x}) \in A$, a $P \cup P'$-formula $\rho_a(\vec{x})$ of the form $\wedge_{p' \in P'} \forall \vec{y}.p'(\vec{y}) \equiv \varphi(\vec{x}\vec{y})$, where φ is a P-formula called the *transition* of a.

We call $M(n, k)$ an $\langle n, k \rangle$ *instance of* M, and we use M to denote the family of instances $\{M(n, k) : n, k \in \mathbb{N}^{>0}\}$.

A *state* s of $M(n, k)$ is an interpretation of the predicates P over the universe U; we use Σ to denote the set of states. A *binding* b to logical variables X is a map from X to U, and $b(x) \in U$ denotes the binding of x. We lift b to vectors of variables, writing $b(\vec{x}) = \vec{u}$ when $|\vec{x}| = |\vec{u}|$ and $b(x_i) = u_i$ for each $0 < i \leq |\vec{x}|$. We denote the empty binding with $\emptyset$. For a P-formula $\varphi(\vec{x})$, a state s, and a binding b to $\vec{x}$, we define the satisfaction relation $s, b \models \varphi$ in the usual way [21]. When φ is a *sentence* (i.e., contains no free variables), we write $s \models \varphi$ instead of $s, \emptyset \models \varphi$. We write $\varphi \models \psi$ when for all s and b such that $s, b \models \varphi$, we have $s, b \models \psi$. We write $\langle s, s' \rangle, b \models \varphi$ when φ is over the language of primed and unprimed predicates and each $p \in P$ is interpreted in s, and each $p \in P' = \{p' : p \in P\}$ is interpreted in s'.

For an action $a(\vec{x})$ and a binding b of $\vec{x}$ such that $\vec{u} = b(\vec{x})$, we say $a(\vec{u})$ is *enabled* in a state s when $s, b \models g_a$, and when $\langle s, s' \rangle, b \models g_a \wedge \rho_a$ we call s' an *$a(\vec{u})$-successor* of s. We say $M(n, k)$ is *deterministic* when for each state s, binding b, and action a, there is at most one $a(\vec{u})$ successor of s, and M is deterministic if $M(n, k)$ is deterministic for each n, k. Let $A_U = \{a(b(\vec{x})) : a(\vec{x}) \in A, b \text{ is a binding of } \vec{x} \text{ to } U\}$. For $w \in A_U^*$, a *w-computation* σ of $M(n, k)$ is a Σ-sequence $s_0 s_1 \ldots$ such that

1. s_0 is initial, i.e., $s_0 \models \Theta$, and
2. for each $\ell = 0, 1, \ldots$, the state $s_{\ell+1}$ is an $a(\vec{u})$-successor of s_ℓ, and $w_\ell = a(\vec{u})$.

The *language* $L(M(n, k))$ of $M(n, k)$ is the set of words w for which there exists a w-computation:

$$L(M(n, k)) = \{w : \exists \sigma.\sigma \text{ is a } w \text{ computation of } M(n, k)\}$$

and we write $L(M) = \bigcup\{L(M(n, k)) : n, k \in \mathbb{N}^{>0}\}$.

A state s is *reachable* in $M(n, k)$ when s occurs in some computation. A sentence φ is an *invariant* of $M(n, k)$ if $s \models \varphi$ for every reachable state s, and φ is *inductive* when $\Theta \models \varphi$ and

$$\varphi \quad \wedge \bigvee_{a(\vec{u}) \in A_U} \exists \vec{u}.g_a(\vec{u}) \wedge \rho_a(\vec{u}) \;\models\; \varphi'.$$

A sentence φ is a *uniform invariant* of M, written $M \models \varphi$, if φ is an invariant of $M(n, k)$ for all $n, k \in \mathbb{N}^{>0}$; in this case we also say that M is *φ-safe*.

Example 1 (Specification of Strict Serializability)**.** A TM specification for strict serializability is a parameterized system $\mathsf{SS}(n, k)$ such that $L(\mathsf{SS}) = L_{SS}$. Guerraoui et al. [16] show a deterministic TM specification for strict serializability. We briefly recall its key components; Appendix D contains the full description.

The set of predicates P consists of the "status" predicates $\mathsf{SS.finished}(t)$, $\mathsf{SS.started}(t)$, $\mathsf{SS.pending}(t)$, $\mathsf{SS.invalid}(t)$, and

$\mathsf{SS.rs}(t, v), \mathsf{SS.ws}(t, v)$	read & write sets
$\mathsf{SS.prs}(t, v), \mathsf{SS.pws}(t, v)$	prohibited-read & -write sets
$\mathsf{SS.wp}(t_1, t_2)$.	weak-predecessor

The predicate $\mathsf{SS.rs}(t, v)$ (resp., $\mathsf{SS.ws}(t, v)$) holds when thread t has read (resp., written) location v in its unfinished transac-

tion. When $\mathsf{SS.prs}(t,v)$ (resp., $\mathsf{SS.pws}(t,v)$) holds, t will become $\mathsf{SS.invalid}$ by reading (resp., writing) location v. When $\mathsf{SS.wp}(t_1,t_2)$ holds, t_1 must *serialize* after the unfinished transaction of t_2 in order for both threads to commit their transactions. As syntactic sugar, we indicate status predicate valuations with atomic formulæ, e.g., by writing $\mathsf{SS.status}(t) = \mathsf{finished}$ to mean

$$\mathsf{SS.finished}(t) \wedge \neg\mathsf{SS.started}(t) \\ \wedge \neg\mathsf{SS.invalid}(t) \wedge \neg\mathsf{SS.pending}(t).$$

The set A_{SS} of actions is $\{read(t,v),\ write(t,v),\ commit(t),\ abort(t)\}$. The initial condition is given by the sentence

$$\forall t_1,t_2,v.\mathsf{SS.status}(t_1) = \mathsf{finished} \wedge \neg\mathsf{SS.wp}(t_1,t_2) \\ \wedge \neg(\mathsf{SS.rs}(t_1,v) \vee \mathsf{SS.ws}(t_1,v) \\ \vee \mathsf{SS.prs}(t_t,v) \vee \mathsf{SS.pws}(t_1,v)).$$

The guards and actions are somewhat complex (see Appendix D), though as an example, for the $commit(t)$ action, the guard is $\neg\mathsf{SS.invalid}(t) \wedge \neg\mathsf{SS.wp}(t,t)$, and the update formula for the $\mathsf{SS.wp}$ predicate is

$$\forall t_1,t_2.\mathsf{SS.wp}'(t_1,t_2) \equiv t_1 \neq t \wedge t_2 \neq t \\ \wedge (\mathsf{SS.wp}(t_1,t_2) \vee \mathsf{SS.wp}(t,t_2) \\ \wedge (\mathsf{SS.wp}(t_1,t) \vee \exists v.\mathsf{SS.ws}(t_1,v) \wedge \mathsf{SS.ws}(t,v))).$$

It is interesting to note that the operational specification of strict serializability is more complex than the TM implementations we consider in this work. Whereas the SS specification does not correspond to an efficient implementation, each actual TM implementation conservatively disallows many serializable concurrent interactions in order to *efficiently* decide which transactions are allowed to commit. Thus we rely on the result of Guerraoui et al. [16] to know that this complex operational specification is correct.

Example 2 (DSTM)**.** We model dynamic software transactional memory [20] as a parameterized system with the status predicates $\mathsf{DSTM.finished}(t)$, $\mathsf{DSTM.validated}(t)$, $\mathsf{DSTM.invalid}(t)$, $\mathsf{DSTM.aborted}(t)$, and

$$\mathsf{DSTM.rs}(t,v), \mathsf{DSTM.os}(t,v). \qquad \text{read \& own sets}$$

The set of actions is $A_{\mathsf{SS}} \cup \{validate(t)\}$. The initial condition is

$$\forall t,v.\mathsf{DSTM.status}(t) = \mathsf{finished} \\ \wedge \neg\mathsf{DSTM.rs}(t,v) \wedge \neg\mathsf{DSTM.os}(t,v).$$

The full specification is given in Appendix B; as an example, the $commit(t)$ action is guarded by $\mathsf{DSTM.validated}(t)$, and updates the $\mathsf{DSTM.validated}$ predicate by the formula

$$\forall t_1.\mathsf{DSTM.validated}'(t_1) \equiv t_1 \neq t \wedge \mathsf{DSTM.validated}(t_1) \\ \wedge \neg\exists v.\mathsf{DSTM.rs}(t_1,v) \wedge \mathsf{DSTM.os}(t,v).$$

Refinement Let $M_1 = \langle U, P_1, A_1, \Theta_1, g_1, \rho_1\rangle$ and $M_2 = \langle U, P_2, A_2, \Theta_2, g_2, \rho_2\rangle$ be two parameterized systems. We say M_1 *is compatible with* M_2 if $A_2 \subseteq A_1$ and $P_1 \cap P_2 = \emptyset$, and we say M_1 *refines* M_2, written $M_1 \preceq M_2$, if $L(M_1(n,k))|_{A_2} \subseteq L(M_2(n,k))$ for every $n,k \in \mathbb{N}^{>0}$, where $L|_A$ is the projection of a language L to the alphabet A.

If $M_1(n,k)$ is compatible with $M_2(n,k)$, we define the *composition* $M_1 \times M_2(n,k) = \langle U, P_1 \cup P_2, A_1, \Theta_1 \wedge \Theta_2, g, \rho\rangle$ of M_1 and M_2 where

$$g_a(\vec{x}) = \begin{cases} g_{1a}(\vec{x}) & \text{if } a \in A_1 \setminus A_2 \\ g_{1a}(\vec{x}) \wedge g_{2a}(\vec{x}) & \text{if } a \in A_2 \end{cases}$$

$$\rho_a(\vec{x}) = \begin{cases} \rho_{1a}(\vec{x}) \wedge \bigwedge_{p\in P_2} p'(\vec{x}) \equiv p(\vec{x}) & \text{if } a \in A_1 \setminus A_2 \\ \rho_{1a}(\vec{x}) \wedge \rho_{2a}(\vec{x}) & \text{if } a \in A_2 \end{cases}$$

Lemma 1. *Let M_1 and M_2 be parameterized systems such that M_1 is compatible with M_2 and M_2 is deterministic. $M_1 \preceq M_2$ iff*

$$M_1 \times M_2 \models \bigwedge_{a(\vec{x})\in A_2} \forall\vec{x}.\ g_{1a}(\vec{x}) \implies g_{2a}(\vec{x}).$$

The proof is a standard reduction from the search for a simulation relation to an invariant check [22]. Since M_2 is deterministic, the condition is both necessary and sufficient [11]. Thus we reduce the TM verification problem to a parameterized model checking problem on the product of the implementation and a deterministic strict serializability model. For example, to show $L(\mathsf{DSTM}) \subseteq L_{SS}$, we construct the deterministic specification SS, and check if

$$\mathsf{DSTM} \times \mathsf{SS} \models \bigwedge_{a(\vec{x})\in A_{\mathsf{SS}}} \forall\vec{x}.\ g_{\mathsf{DSTM},a}(\vec{x}) \implies g_{\mathsf{SS},a}(\vec{x}).$$

It turns out that the guards for all actions of SS except for $commit(t)$ are $true$, and the guard for the $commit(t)$ action in SS and DSTM are respectively,

$$\neg\mathsf{SS.invalid}(t) \wedge \neg\mathsf{SS.wp}(t,t), \text{ and } \mathsf{DSTM.validated}(t).$$

Thus the refinement verification question systematically reduces to

$$\mathsf{DSTM} \times \mathsf{SS} \models \forall t.\mathsf{DSTM.validated}(t) \\ \implies \neg\mathsf{SS.invalid}(t) \wedge \neg\mathsf{SS.wp}(t,t). \quad (1)$$

This invariant verification problem still is a difficult one, since there is no bound on the parameters (i.e., the number of threads n and locations k), and the transition relations for SS and DSTM use universal or existential quantification over threads and locations.

4. Parameterized Safety Verification

Here we present a technique to check if an first-order sentence ψ is a uniform invariant of a parameterized system M. Our method is based on the deductive safety verification rule [25]:

$$\frac{\begin{array}{ll} I_1: & \Theta \models \varphi \\ I_2: & \varphi \wedge \bigvee_{a(\vec{x})\in A} \exists\vec{x}.g_a \wedge \rho_a \models \varphi' \\ I_3: & \varphi \models \psi \end{array}}{M \models \psi} \quad (\textsc{Inv})$$

To show that ψ is a uniform invariant of M, we use an auxiliary inductive invariant φ such that (I_1) the initial condition Θ of M satisfies φ, (I_2) transitions of M preserve φ, and (I_3) φ is stronger than ψ. Conditions I_1 and I_2 imply φ is a uniform inductive invariant of M. The steps of deductive verification are (i) generating the inductive invariant φ, and (ii) validating that φ satisfies Conditions I_1, I_2, and I_3. In our approach, we solve these issues as follows:

(i) Invariant Generation We generate candidate invariants by instantiating formulæ from a fixed template language. We then performing reachability analysis on instances $M(n,k)$ of the parameterized family for small fixed values of n,k. Candidates which are not invariants of the small instances are discarded. Since $M(n,k)$ is finite, the reachability analysis terminates. The *candidate inductive invariant* is the conjunction, over all formulæ instantiated from the template, of the formulæ that are invariants of $M(n,k)$ (i.e., that contain the set of reachable states of $M(n,k)$).

(ii) Invariant Validation Since our formulæ use universal and existential quantifiers, checking each step of the deductive safety verification rule INV is undecidable. Rather than develop our own quantifier instantiation heuristics, we choose to rely on highly engineered theorem provers to efficiently discharge each implication query.

Input : Parameterized system $M(n,k)$; initial bounds n_0, k_0, j_0; formula ψ, template enumeration TL.
Result: Whether or not $M \models \psi$.

```
1  ⟨n, k⟩ ← ⟨n_0, k_0⟩; j ← j_0; Invariant ← true;
2  repeat
       // 1a. Explore the bounded instance
3      rs ← Reach(M(n, k));
4      if rs ⊭ ψ then
5          return "M is not ψ-safe";
6      end
       // 1b. Discard non-invariants
7      candidates ← {φ : φ generated by TL[j] and rs ⊨ φ};
       // 2. Candidate invariant weakening
8      recheck ← true;
9      while candidates ≠ ∅ and recheck do
10         recheck ← false;
11         foreach φ ∈ candidates do
12             let ξ_{a(x⃗)} = Invariant ∧ ⋀ candidates
13                           ∧ ∃x⃗.(g_a ∧ ρ_a);
14             if exists a(x⃗) ∈ A s.t. not Check( ξ_{a(x⃗)} ⊨ φ' )
15             or not Check( Θ ⊨ φ ) then
16                 candidates ← candidates \ {φ};
17                 recheck ← true;
18             end
19         end
20     end
       // 3. Check invariant strength
21     Invariant ← Invariant ∧ ⋀ candidates;
22     if Check( Invariant ⊨ ψ) then
23         return "M is ψ-safe"
24     end
       // Retry with bigger bounds
25     increment n, k, j;
26 end
```

Algorithm 1: Parameterized safety verification.

Our parameterized verification technique is summarized by Algorithm 1. It takes as input a parameterized system M and a property ψ. The parameters n_0 and k_0 range over the number of threads and locations in finite instances, and the parameter j_0 is used to as an initial template enumeration index.

The algorithm has three phases. The first phase (lines 3–7) generates a set of candidate invariants. The set of reachable states rs of a finite instance $M(\mathsf{n},\mathsf{k})$ are computed (line 3); if some reachable state violates ψ (lines 4–6), ψ cannot be a uniform invariant of M, and a safety violation is reported. Otherwise, a set of candidates is generated by choosing each formula in the current invariant template $\mathsf{TL}[\mathsf{j}]$ that contains the set of reachable states (line 7).

The second "candidate invariant weakening" phase (on lines 8–20) discards candidates that can not be proved inductive, either because they are not preserved by some transition (line 14), or because they are not initially valid (line 15). The conjunction of candidates remaining at the end of the loop of lines 8–20 is a uniform inductive invariant.

In the third phase the inductive invariant strength is checked. If the current invariant implies ψ, then by the INV rule, ψ is a uniform invariant. Otherwise, we increase the indexes n, k, and j, and repeat the top-level loop of line 2.

It is easy to see that Algorithm 1 is sound. However, since it is impossible to know the size (i.e., the number of quantifiers and literals) of the required candidate invariants, there is no bound on the template enumeration index j, and thus there is no telling when the procedure may terminate. Supposing that the required invariants are expressible in first-order logic, and the `Check` routine is complete (though in actuality it is not), some template is guaranteed to capture the required invariant, and termination would be eventually guaranteed.

5. Template-Based Candidate Generation

We generate candidate invariants following the idea of *verification by invisible invariants* [2, 28]. We compute the set of reachable states of $M(n,k)$ for fixed, finite thread- and location-count values of n and k; this is possible since the state space of $M(n,k)$ is finite. A candidate uniform invariant for M is proposed by generalizing invariants for $M(n,k)$.

We depart from the technique of Arons et al. [2] to generate candidate invariants for two reasons. First, the generated invariant can be too specific. For example, let R be the set of reachable states of $M(n,k)$. The synthesized invariant of Arons et al. [2] has the form $\forall\vec{x}.\bigvee_{s\in R}\psi_s(\vec{x})$ where each ψ_s is a minterm corresponding to a *complete* interpretation of the predicates P on the variables $\vec{x}$ (i.e., for every predicate $p \in P$ and sublist $\vec{y}$ of variables in $\vec{x}$ with the proper sorts, either $p(\vec{y})$ or $\neg p(\vec{y})$ appears in ψ_s). Symmetries between distinct minterms—for example, minterms which are identical up to variable renaming—are not exploited, leading to much redundancy and overly-specific invariants. Checking the validity of such a large invariant is expensive. Second, the invariant is often too specific to n and k to be a uniform invariant as a whole. However, if a (monolithic) candidate invariant derived from $M(n,k)$ is not a uniform invariant, there is no clear way to proceed to generate weaker candidate invariants.

Example 3. Let $M = \mathsf{DSTM}\times\mathsf{SS}$. Although the sentence

$$\begin{aligned}\forall t,v.(&\mathsf{SS.pending}(t) \Rightarrow \mathsf{SS.wp}(t,t))\\ &\wedge (\mathsf{DSTM.rs}(t,v) \Rightarrow \neg\mathsf{SS.pending}(t))\\ &\vee \neg\mathsf{DSTM.aborted}(t)\end{aligned} \tag{2}$$

is an invariant of $M(2,1)$, it is not an invariant of $M(3,2)$, and hence not a uniform invariant. However, consider the invariants

$$\begin{aligned}&\forall t.\mathsf{SS.pending}(t) \wedge \mathsf{DSTM.aborted}(t) \Rightarrow \mathsf{SS.wp}(t,t),\\ &\forall t,v.\mathsf{SS.pending}(t) \wedge \mathsf{DSTM.aborted}(t)\\ &\qquad\Rightarrow \neg\mathsf{DSTM.rs}(t,v)\end{aligned} \tag{3}$$

whose conjunction is logically equivalent to Equation 2. Both are invariants of $M(2,1)$, though only the second is uniform. Here, even though the conjunction it not uniform, we can extract a weaker candidate (i.e., the second conjunct) that is.

Example 3 indicates that we should look for *conjunctive* candidate invariants. Algorithm 1 (lines 8-20) shows how conjunctive invariants can be individually checked for inductiveness and weakened by discarding conjuncts that are not inductive. We restrict the shape of the extracted invariants by imposing template-based invariant generation [8, 32] while generating candidate invariants.

Let Φ be a finite set of *symbolic predicate variables*. A *template* τ is a Φ-sentence in prenex form, and a *template schema* T is a finite set of templates. Let Ψ be a set of atomic formulas whose free variables are a subset of the quantified variables of τ. A sentence φ is *generated by* template τ when there exists a binding f from the predicate variables Φ to atomic formulas in Ψ such that the formula $\tau[f(\Phi)/\Phi]$ obtained by replacing each $\phi \in \Phi$ with $f(\phi)$ is syntactically equal to φ. We say φ is *generated by* template schema T when φ is generated by some $\tau \in T$.

To generate candidate invariants, we fix a template schema T and fix the set of atomic formulæ $\Psi = \{p(\bar{x}), \neg p(\bar{x}) \mid p \in P\} \cup \{true\}$. Then, for each φ generated from T, we check if φ is an invariant of $M(n,k)$. If so, it is added as a conjunct of the candidate invariant; otherwise it is discarded. This is shown in line 7 of Algorithm 1.

In practice we restrict the template schema to *small* templates, since enumerating all formulas generated by a template is expensive. We found that templates of the form $\forall t, v. \phi_1 \wedge \phi_2 \Rightarrow \phi_3$ sufficed to generate sufficiently strong uniform invariants.

If a necessary uniform invariant cannot be generated by a template schema, then the verification will fail. In this case, Algorithm 1 achieves relative completeness by incrementing the template enumeration index j; for every first-order formula φ there is some j such that φ is generated by $\mathsf{TL}[\mathsf{j}]$ (lines 7 and 25). In practice, incrementing j has not been necessary.

Example 4. Using the template schema $\forall t, v. \phi_1 \wedge \phi_2 \Rightarrow \phi_3$ and the set of atomic formulas $\{p(\bar{v}), \neg p(\bar{v}) \mid p \in P_{\mathsf{SS}} \cup P_{\mathsf{DSTM}}\} \cup \{true\}$, we generated 525 candidate invariants from the $\langle 2,1 \rangle$-instance of $\mathsf{DSTM} \times \mathsf{SS}$; 341 of these proved to be uniform. The found invariants include:

$$\forall t, v. \mathsf{SS.rs}(t,v) \wedge \neg \mathsf{DSTM.aborted}(t) \Rightarrow \mathsf{DSTM.rs}(t,v)$$
$$\forall t, v. \mathsf{SS.ws}(t,v) \wedge \neg \mathsf{DSTM.aborted}(t) \Rightarrow \mathsf{DSTM.os}(t,v)$$
$$\forall t. \mathsf{DSTM.validated}(t) \Rightarrow \neg \mathsf{SS.wp}(t,t)$$
$$\forall t. \mathsf{DSTM.validated}(t) \Rightarrow \neg \mathsf{SS.invalid}(t)$$
$$\forall t. \mathsf{DSTM.validated}(t) \Rightarrow \neg \mathsf{SS.pending}(t).$$

Notice that the refinement verification obligation of Equation 1 is implied by the conjunction of these invariants.

6. Implementation and Experimental Evaluation

We have implemented our technique by extending the TVLA framework [24] to compute the reachable state space of bounded instances of the input systems and extract quantified candidate invariants. The candidate invariants are subsequently checked using the SPASS automated theorem prover [34].

Our models of the two-phase locking (TPL), dynamic software transactional memory (DSTM), and transactional locking II (TL2) implementations, along with the deterministic strict serializability specification, are derived from the parameterized automaton models provided by Guerraoui et al. [15, 16]. The models are listed in full in Appendices A–D.

Since the SS specification remains fixed, we use our technique to initially generate a uniform inductive invariant of SS once-and-for-all, before considering the implementation and specification product systems. We then use this invariant to strengthen the invariants of each product system.

Experimental setting For our experimental evaluation we use the fixed template schema $\forall x, y. \phi_1 \wedge \phi_2 \Rightarrow \phi_3$ to generate every possible three-term formula over two universally quantified variables—modulo typing concerns, and obvious syntactic symmetries, e.g., $\forall x. p(x) \wedge q(x) \Rightarrow r(x)$ is logically equivalent to $\forall x. q(x) \wedge p(x) \Rightarrow r(x)$ and $\forall x. p(x) \wedge \neg r(x) \Rightarrow \neg q(x)$.

Following Algorithm 1, we perform an initial filtering to discard the candidate invariants φ which are not invariants of a small system instance $M(n_0, k_0)$ by finding a reachable state of $M(n_0, k_0)$ which does not satisfy φ. To further reduce the number of candidates, we discard invariants that are redundant with—i.e., implied by the conjunction of—the known system constraints and previously-extracted candidates. For example, constraints on the DSTM system dictate that a single thread cannot be both DSTM.validated and DSTM.aborted, so the invariant

$$\forall t, v.\ \mathsf{DSTM.validated}(t) \wedge \neg \mathsf{DSTM.rs}(t,v) \Rightarrow \neg \mathsf{DSTM.aborted}(t)$$

is redundant w.r.t. system constraints, while the uniform invariant of Equation 3 is redundant w.r.t. the stronger invariant

$$\forall t, v. \mathsf{DSTM.aborted}(t) \Rightarrow \neg \mathsf{DSTM.rs}(t,v).$$

Due to scalability limitations of our inefficient implementation, we did not check redundancy w.r.t. *all* previously generated candidates. Instead, we split the generated candidates into 40 partitions, and only redundancy w.r.t. candidates of the same partition (and system constraints) is considered. Note that *full* redundancy elimination may sometimes drop potential uniform invariants (and consequently fail to verify a system), since a uniform invariant may be considered redundant w.r.t. to a simpler non-uniform candidate. For example, suppose the predicates $x = 0$ and $x \geq 0$ are both invariants of a finite instance, but only $x \geq 0$ is a uniform invariant. By eliminating $x \geq 0$ because it is redundant w.r.t. $x = 0$, we may not find a strong enough uniform invariant.

The second phase of our implementation checks inductiveness of the remaining candidate invariants. In each *weakening* iteration we attempt to validate each candidate via theorem prover with a 10 second time limit. When a candidate cannot be validated—either due to non-uniformity, or prover incompleteness/timeout—it is discarded. At the end of a weakening iteration where no candidates were discarded, we conclude that the conjunction of candidate invariants is inductive. When a weakening iteration finishes after discarding some candidate, we are obliged to start another iteration to ensure that the discarding of one candidate does not invalidate another.

Table 1 lists our experimental measurements. For each small instance of each system, we record the number of reachable states and the time spent exploring them in the second column group ("bounded exploration"). The third column group ("template instantiations") lists (1) the *total* number of formulæ generated from templates (column "total"), (2) the number of candidate formulæ from (1) which were *invariants* of the given instance (column "invs."), (3) the number of *non-redundant* candidates from (2) (column "non-red."), and (4) the number of non-redundant candidates from (3) that were finally *validated* as uniform invariants in the subsequent validation and invariant weakening steps (column "val."). The fourth column group (column "iter") lists the number of invariant weakening iterations, and finally the last column group ("validation time") lists the average *per-candidate* validation time, and the *total* runtime, including validating (or refuting) each candidate for each weakening iteration.

Discussion In all cases except TPL, our crude prototype implementation only scales to systems with at most two threads and two memory locations. For larger instances our tool runs for several minutes before exhausting a 2GB memory limit. Based on the system instances we did explore, varying the number of memory locations has little effect on the system behavior, though varying the number of threads has a noticeable effect: only about half of the invariants of one-thread DSTM and TL2 systems are invariants of the two-thread systems. It would be interesting to measure how invariants vary with more than two threads using a more scalable tool. In the case of the simplistic TPL system, it is clear that all possible system behaviors between two threads are expressed with only two threads, since every $\langle 2,1 \rangle$-invariant is uniform.

Although not strictly required, filtering candidate formulæ that are either not invariants on the bounded instance, or are redundant with other candidates, can greatly reduce the runtime of the subsequent invariant validation step. For example, in the $\langle 2,1 \rangle$ instance of $\mathsf{DSTM} \times \mathsf{SS}$, only 525 of 14,724 candidates survived the filtering.

system	$\langle n,k\rangle$	bounded exploration states	bounded exploration time	template instantiations total	template instantiations invs.	template instantiations non-red.	template instantiations val.	iter.	validation time per-cand.	validation time total
SS	$\langle 1,1\rangle$	4	2.5s	|	2,542	780	346	6	0.38s	35m45s
	$\langle 1,2\rangle$	16	3.7s	3,520	2,542	780	346	6	0.38s	35m26s
	$\langle 2,1\rangle$	74	5.7s	|	1,313	432	377	3	0.65s	19m0s
	$\langle 2,2\rangle$	7296	3m28s	|	1,237	402	377	3	0.42s	12m29s
TPL × SS	$\langle 1,1\rangle$	4	10.5s	|	3,344	434	430	2	0.38s	7m28s
	$\langle 1,2\rangle$	16	8.0s	|	|	|	|	|	0.41s	8m38s
	$\langle 2,1\rangle$	8	8.5s	|	3,330	430	430	1	0.38s	4m5s
	$\langle 2,2\rangle$	64	14.0s	4,924	|	|	|	|	0.39s	4m15s
	$\langle 2,3\rangle$	512	43.7s	|	|	|	|	|	0.46s	5m13s
	$\langle 3,2\rangle$	196	20.5s	|	|	|	|	|	0.44s	4m56s
	$\langle 3,3\rangle$	2,744	3m26s	|	|	|	|	|	0.41s	4m25s
DSTM × SS	$\langle 1,1\rangle$	8	5.4s	|	9,806	1,230	298	6	0.92s	71m10s
	$\langle 1,2\rangle$	32	9.1s	14,724	9,806	1,230	298	6	0.74s	55m15s
	$\langle 2,1\rangle$	184	9.7s	|	4,512	525	341	3	1.71s	38m10s
	$\langle 2,2\rangle$	15.6K	5m6s	|	4,308	453	346	3	0.97s	21m56s
TL2 × SS	$\langle 1,1\rangle$	6	3.6s	|	10,068	1178	–‡	–	–	–
	$\langle 1,2\rangle$	36	5.4s	14,706	10,068	1,178	–‡	–	–	–
	$\langle 2,1\rangle$	344	13.4s	|	4,695	556	416	2	5.06s†	100m4s†
	$\langle 2,2\rangle$	100K	55m21s	|	–*	–	–	–	–	–

Table 1. Transactional memory verification experiments. The parameters n and k indicate the number of threads and memory locations per instance. †The per-candidate theorem prover timeout was set to 30s. ‡The candidate invariants could not be proved inductive. *Invariant extraction did not scale here.

Assuming the same 1.71s average time[1] per validation step, the total validation time without filtering the candidates would have been at least seven hours, rather than only 38 minutes. Similarly, the size of the initial bounded instance can seriously affect the validation time by reducing the number of candidates. For instance, 5,294 (705 non-redundant) quantified invariants of the $\langle 1,1\rangle$ DSTM × SS instance are not invariants of the $\langle 2,1\rangle$ instance, thus cannot be uniform invariants, and can be filtered from the concrete instance in a few seconds. Note that the reachability computation (i.e., bounded exploration) can be soundly terminated at any point, effectively shifting the validation effort toward the theorem prover back-end.

It should also be noted that this redundant candidate elimination scheme can result in somewhat unpredictable behavior. In the case of TL2, discarding candidate invariants extracted from the $\langle 1,1\rangle$ and $\langle 1,2\rangle$ instances that are redundant with previously extracted candidates failed to produce an uniform inductive candidate subset. The candidates extracted from the $\langle 2,1\rangle$ instance were sufficient, though the our extraction tool was unable to process the large number of reachable states in the $\langle 2,2\rangle$ instance.

Although we did not implement more intricate redundant candidate elimination schemes, there are several possibilities. One idea is to *temporarily* discard each redundant candidate φ, while recording the candidates $\{\psi_i\}_i$ rendering φ redundant. When some ψ_i is subsequently invalidated, φ can be reconsidered since it is no longer redundant. This scheme would very likely reduce the overall runtime of verification for TL2 × SS. We leave all such strategies for future work.

In only two cases did our very simple template schema not produce strong enough candidate invariants. An important invariant of SS states that weak-predecessors of a thread t that are not SS.invalid or SS.pending have a read-write conflict with t, e.g.,

$$\forall t_1, t_2.\mathsf{SS.wp}(t_1,t_2) \wedge \mathsf{SS.finished}(t_2) \\ \Rightarrow \exists v.\mathsf{SS.rs}(t_2,v) \wedge \mathsf{SS.ws}(t_1,v).$$

[1] Though in actuality the time spent validating each candidate increases as the number of candidates increases.

Second, a necessary invariant of TL2×SS states that a thread which is both TL2.finished and SS.invalid or SS.pending must have some memory location in both its read set and modified set. Although these invariants can be captured by the template schema

$$\forall x, y.\phi_1 \wedge \phi_2 \Rightarrow \exists z.\phi_3 \wedge \phi_4,$$

due to scalability limitations of our crude prototype implementation, we prefer instead to augment the systems with the additional predicates $\mathsf{SS.rwc}(t_1,t_2)$ and $\mathsf{TL2.rmc}(t)$ to encode "read-write conflicts" in SS, and "read-modified conflicts" in TL2. These additional predicates, along with the template schema fixed above, allow us to express all the required invariants for our proofs.

Finally, probably due to the greater complexity of the TL2 system, we found it necessary to extend the 10s theorem prover timeout to 30s in the candidate validation step.

Bug finding In addition to the verification of these models, our technique was able to find (from a $\langle 2,1\rangle$-size instance in 6.4 seconds) the known bug [15] in a TL2 implementation which swaps the order of the *lock* and *validate* actions: the trace

$$write(t_1,v)\ write(t_2,v)\ read(t_2,v)\ read(t_1,v) \\ validate(t_1)\ validate(t_2)\ lock(t_2,v) \\ commit(t_2)\ lock(t_1,v)\ commit(t_1)$$

is allowed by this alternate implementation, though is not strictly serializable.

Limitations Although here we restrict the invariant language to first-order logic, we have found that expressing inductive invariants of certain TMs requires transitive closure. The TL2 model we've verified executes the *validate* and *commit* actions atomically, though an implementation performing these actions separately seems to require the invariant "there are no SS.wp-cycles involving only TL2.validated transactions."

The systems we verify are modeled using high-level *abstract data types*, e.g., the read- and write-sets are modeled as *mathematical sets* rather than a low-level heap encoding of sets. Although real-world software TM implementations are indeed programmed

using linked data structures, establishing the link between ADT specifications and ADT implementations is an orthogonal problem and is better understood [29, 36].

7. Related Work

Our starting point is the formalization of transactional memory implementations, and the strict serializability specification, as finite-state automata by Guerraoui et al. [15]. Even with these models, *automatic* parameterized verification for transactional memory systems remained beyond the ability of existing verification tools.

The problem of verifying transactional memories has been studied before [6, 7, 15–17, 33]. Cohen et al. [6] verified refinement mappings between small TM instances and a strict serializability specification by explicit state model checking; they subsequently extended their approach to parameterized TM verification via an interactive proof assistant [7]. In both cases the proofs required manually prescribing the refinement mappings—a task which requires familiarity with the implementation, the specification, and the relationship between the two. Taşiran [33] took a similar approach, using existing software verification tools to verify TM implementations which were manually annotated with pre- and post-conditions corresponding to an insightful proof decomposition. In contrast, our use of intermediate invariants is *invisible* to the user, and the resulting proof is automatic.

The approach of Guerraoui et al. [15, 16, 17] is based on a small model theorem reducing parameterized TM verification to verification on an instance with two threads and two memory locations; their theorem applies only to systems satisfying certain conditions, some of which are easy to check syntactically (e.g., thread symmetry), and others which would be difficult to check for an arbitrary implementation, and seem to require human insight. In contrast, our proof technique is sound for any transactional memory implementation—regardless of the conditions of Guerraoui et al. [15], except for symmetry—but like any heuristic, may fail to verify a given implementation. Our experimental results demonstrate that our approach does succeed for common implementations.

Our experiments with existing software verification tools fell short. Software model checkers based on predicate abstraction and counterexample-guided abstraction refinement, such as SLAM [3] and BLAST [18], typically implement a very coarse memory model (e.g., based on an imprecise alias analysis) which is not sufficiently detailed to reason about the interactions between arbitrarily many threads and memory locations. In our experiments, the BLAST model checker was unable to verify even finite instances of the implementations because of a coarse modelling of arrays. In addition, the default counterexample refinement procedure in BLAST, which looks individually at abstract counterexamples, could not infer the universally quantified invariants that were required in the proof, instead entering an infinite refinement loop.

Tools based on three-valued shape analysis [31] or separation logic [30], such as TVLA [24] and SPACEINVADER [35], can theoretically prove properties of programs manipulating unbounded heaps. In practice, however, their most successful applications have been in proving complex data structure invariants in small code, or simple data structure invariants in large code. Our attempts to verify transactional memory using TVLA failed for two reasons: first, it required intensive manual interactions to identify the instrumentation predicates necessary to rule out abstract counterexamples, and second, the tool exhausted our time and space limitations before finishing, even with a small number of instrumentation predicates.

We use and extend the idea of *verification by invisible invariants* [2, 28] from the parameterized hardware and protocol verification setting. Instead of generating invariants with arbitrary Boolean structure, we combine the idea with template-based invariant generation to restrict invariants to the simple form of template formulæ conjunctions. This ensures that even when the candidate formula *as a whole* is not an invariant, useful *parts* of the formula can be salvaged. In contrast, if the inductiveness check fails for a arbitrarily structured Boolean formula, it is not clear how invalid parts of it can be discarded. Our original attempt to synthesize inductive invariants from concrete executions produced formulas containing hundreds, or thousands, of cubes which we were not able to reduce. Various optimization techniques, e.g., using BDDs, did not help. With templates we manage the size and complexity of potential invariants, and can bias the search for candidate invariants using observations made for the class of systems we consider. While template-based invariant generation has been studied before, applications so far have been limited to arithmetic constraints [5, 8, 32]. In contrast, our templates range over quantified predicates relating the data structures of concurrently executing threads.

The initial filtering of candidate invariants on small system instances can be seen as a variation of *dynamic detection of likely invariants* [14], though there the initial candidate set is restricted to much simpler formulæ (e.g., whether or not a pointer-value can be null) or user-specified candidates.

Our idea of exploiting thread and memory location symmetry is inspired by techniques from parameterized safety proofs in hardware verification [26]. However, while the proof obligations in that hardware setting after symmetry reduction were essentially Boolean problems, we have generalized the techniques to incorporate expressive first-order theories. Besides the increase in expressive power, this enables us to reason about transition relations and inductive invariants including both universal and existential quantifiers. In contrast, the proof arguments of McMillan [26] require manually specification of witnesses for existential formulæ explicitly by including them as auxiliary state in the models.

Berdine et al. [4] extend three-valued shape analysis to reduce reasoning about parameterized concurrent programs to the behavior of individual threads. However, their technique does not handle existentially quantified properties and requires manually specified instrumentation predicates.

8. Conclusion

Although we have demonstrated the automatic verification of transactional memory systems with respect to strict serializability, there are several additional complications to consider in real-world implementations. Some systems allow nested transactions, or non-transactional shared memory accesses, and may be running on top of weak memory models. Also, the actual implementations use heap-based data structures; analysis generally requires showing a correspondence between the abstract data types (e.g., read- and write-*sets*) and the underlying implementations. Additional safety (e.g., opacity) and liveness properties (e.g., obstruction freedom) are also expected to hold, and should be verified.

Another interesting direction is to apply our template-based algorithm to other systems, such as cache coherence protocols.

Acknowledgments

We would like to thank the anonymous referees, and Amit Goel, Ranjit Jhala, and Todd Millstein for their helpful comments.

References

[1] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[2] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, pages 221–234, 2001.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[4] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, pages 399–413, 2008.

[5] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, pages 378–394, 2007.

[6] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *FMCAD*, pages 37–44, 2007.

[7] A. Cohen, A. Pnueli, and L. D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *CAV*, pages 121–134, 2008.

[8] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

[9] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[11] D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for language inclusion using simulation preorders. In *CAV*, pages 255–265, 1991.

[12] A. Dragojević, R. Guerraoui, and M. Kapałka. Dividing transactional memories by zero. In *TRANSACT*, 2008.

[13] B. Dutertre and L. de Moura. The YICES SMT solver. `http://yices.csl.sri.com`.

[14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[15] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *PLDI*, pages 372–382, 2008.

[16] R. Guerraoui, T. A. Henzinger, and V. Singh. Completeness and nondeterminism in model checking transactional memories. In *CONCUR*, pages 21–35, 2008.

[17] R. Guerraoui, T. A. Henzinger, and V. Singh. Software transactional memory on relaxed memory models. In *CAV*, pages 321–336, 2009.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[20] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[21] S. Kleene. *Introduction to Metamathematics*. North Holland, 1980.

[22] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[23] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[24] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.

[25] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[26] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999.

[27] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[28] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97, 2001.

[29] J. Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005.

[30] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[31] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[32] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234, 2009.

[33] S. Taşiran. A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research, April 2008.

[34] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *CADE*, pages 140–145, 2009.

[35] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, pages 385–398, 2008.

[36] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI*, pages 349–361, 2008.

A. Model of Two-Phase Locking

The two-phase locking system is a transaction manager which ensures sequential behavior by associating two separate locks to each memory location. Read-access is granted to any number of threads so long as no thread has write-access, while any thread with write-access to a location is guaranteed exclusive access. Our TPL model uses the predicates $\mathsf{TPL.rs}(t, v)$ and $\mathsf{TPL.ws}(t, v)$ to mean that t has read or write access to v. The initial condition

$$\forall t, v. \neg \mathsf{TPL.rs}(t, v) \wedge \neg \mathsf{TPL.ws}(t, v)$$

asserts that all access sets are uninhabited. The system is further constrained by an invariant stating that only one thread is allowed write access at any given time:

$$\forall t_1, t_2, v. \mathsf{TPL.ws}(t_1, v) \wedge \mathsf{TPL.ws}(t_2, v) \Rightarrow t_1 = t_2.$$

Although in principle this constraint can be encoded in the transition relation, we state it here to avoid redundancy. This invariant should be considered implicitly, without the need for validation, since it corresponds to the implementation strategy of associating a write-lock per location.

Read The $read(t, v)$ action for TPL is guarded by

$$\forall t_1. \mathsf{TPL.ws}(t_1, v) \Rightarrow t_1 = t,$$

and the transition formula is given by the conjunction of

$$\begin{aligned}&\forall t_1, v_1. \mathsf{TPL.rs}'(t_1, v_1) \equiv \\ &\qquad \mathsf{TPL.rs}(t_1, v_1) \vee \neg \mathsf{TPL.ws}(t, v) \wedge t_1 = t \wedge v_1 = v \\ &\forall t_1, v_1. \mathsf{TPL.ws}'(t_1, v_1) \equiv \mathsf{TPL.ws}(t_1, v_1).\end{aligned}$$

In other words, t can read v when no other thread has write-access, in which case v is added to the read-set of t if v is not already in the write set of t. From this point on we will omit writing identity predicate update formulæ of the form $\forall \vec{x}. p'(\vec{x}) \equiv p(\vec{x})$; its presence will be implicit whenever the update formula for p is omitted.

Write The $write(t, v)$ action for TPL has the guard

$$\forall t_1. \mathsf{TPL.rs}(t_1, v) \vee \mathsf{TPL.ws}(t_1, v) \Rightarrow t_1 = t,$$

and the transition formula is given by

$$\forall t_1, v_1. \mathsf{TPL.ws}'(t_1, v_1) \equiv \mathsf{TPL.ws}(t_1, v_1) \vee t_1 = t \wedge v_1 = v.$$

In other words, t can write to v when no other thread has read- or write-access. In that case, v is added to the write set of t.

Commit & Abort The $commit(t, v)$ and $abort(t, v)$ actions for TPL are always enabled (i.e., they have the guard $true$), and their transition formulæ are given by the conjunction of

$$\begin{aligned}&\forall t_1, v_1. \mathsf{TPL.rs}'(t_1, v_1) \equiv \mathsf{TPL.rs}(t_1, v_1) \wedge t_1 \neq t \\ &\forall t_1, v_1. \mathsf{TPL.ws}'(t_1, v_1) \equiv \mathsf{TPL.ws}(t_1, v_1) \wedge t_1 \neq t.\end{aligned}$$

Both the $commit$ and $abort$ action for TPL are essentially no-ops; the read- and write-sets of the acting thread are simply cleared.

B. Model of Dynamic Software Transactional Memory

The DSTM system is a transaction manager which ensures perceived sequential behavior by allowing only one transaction write-access to a memory location, and ensuring that all values read by a transaction have not been altered by other threads before committing. Our DSTM model uses the thread-status predicates

$$\mathsf{DSTM.finished}(t), \mathsf{DSTM.validated}(t),$$
$$\mathsf{DSTM.invalid}(t), \mathsf{DSTM.aborted}(t),$$

as well as the predicates $\mathsf{DSTM.rs}(t, v)$ and $\mathsf{DSTM.os}(t, v)$ to mean that t has read- or ownership- access to v. As syntactic sugar, we indicate status predicate valuations with atomic formulæ, e.g., by using the formula $\mathsf{DSTM.status}(t) = \mathsf{finished}$ to mean

$$\mathsf{DSTM.finished}(t) \wedge \neg\mathsf{DSTM.validated}(t)$$
$$\wedge \neg\mathsf{DSTM.invalid}(t) \wedge \neg\mathsf{DSTM.aborted}(t).$$

The initial condition

$$\forall t.\mathsf{DSTM.status}(t) = \mathsf{finished}$$
$$\wedge \forall t, v.\neg\mathsf{DSTM.rs}(t, v) \wedge \neg\mathsf{DSTM.os}(t, v)$$

asserts that all access sets are initially uninhabited, and threads begin in the finished state. The system is further constrained by the invariants $\forall t.$ exactly one of

$$\mathsf{DSTM.finished}(t), \mathsf{DSTM.validated}(t), \mathsf{DSTM.invalid}(t),$$
$$\mathsf{DSTM.aborted}(t)$$

holds, and

$$\forall t_1, t_2, v.\mathsf{DSTM.os}(t_1, v) \wedge \mathsf{DSTM.os}(t_2, v) \Rightarrow t_1 = t_2.$$

The first constraint is an artifact of encoding the four-phase DSTM state with predicates, and the second corresponds to the implementation strategy of associating a write-lock per location.

Read The $read(t, v)$ action for DSTM has the guard

$$\neg\mathsf{DSTM.aborted}(t) \wedge (\neg\mathsf{DSTM.os}(t, v) \Rightarrow \mathsf{DSTM.finished}(t)),$$

and the transition formula given by

$$\forall t_1, v_1.\mathsf{DSTM.rs}'(t_1, v_1) \equiv$$
$$\mathsf{DSTM.rs}(t_1, v_1) \vee \neg\mathsf{DSTM.os}(t, v) \wedge t_1 = t \wedge v_1 = v.$$

In other words, any thread t that has not aborted can read v so long as t is in the finished state, or already owns v. If t does not already own v, then v is added to t's read set.

Write The $write(t, v)$ action has the guard $\neg\mathsf{DSTM.aborted}(t)$ and the transition formula given by the conjunction of

$$\forall t_1.\mathsf{DSTM.finished}'(t_1) \equiv$$
$$\mathsf{DSTM.finished}(t_1) \wedge (\neg\mathsf{DSTM.os}(t, v) \Rightarrow \neg\varphi^{t,v}_{\mathrm{abt}}(t_1))$$
$$\forall t_1.\mathsf{DSTM.validated}'(t_1) \equiv$$
$$\mathsf{DSTM.validated}(t_1) \wedge (\neg\mathsf{DSTM.os}(t, v) \Rightarrow \neg\varphi^{t,v}_{\mathrm{abt}}(t_1))$$
$$\forall t_1.\mathsf{DSTM.invalid}'(t_1) \equiv$$
$$\mathsf{DSTM.invalid}(t_1) \wedge (\neg\mathsf{DSTM.os}(t, v) \Rightarrow \neg\varphi^{t,v}_{\mathrm{abt}}(t_1))$$
$$\forall t_1.\mathsf{DSTM.aborted}'(t_1) \equiv$$
$$\mathsf{DSTM.aborted}(t_1) \vee \neg\mathsf{DSTM.os}(t, v) \wedge \varphi^{t,v}_{\mathrm{abt}}(t_1)$$
$$\forall t_1, v_1.\mathsf{DSTM.rs}'(t_1, v_1) \equiv$$
$$\mathsf{DSTM.rs}(t_1, v_1) \wedge (\neg\mathsf{DSTM.os}(t, v) \Rightarrow \neg\varphi^{t,v}_{\mathrm{abt}}(t_1))$$
$$\forall t_1, v_1.\mathsf{DSTM.os}'(t_1, v_1) \equiv$$
$$(\mathsf{DSTM.os}(t, v) \Rightarrow \mathsf{DSTM.os}(t_1, v_1))$$
$$\wedge (\neg\mathsf{DSTM.os}(t, v) \Rightarrow$$
$$(t_1 = t \wedge v_1 = v \vee \mathsf{DSTM.os}(t_1, v_1))$$
$$\wedge \neg\varphi^{t,v}_{\mathrm{abt}}(t_1))$$

where the sub-formula $\varphi^{t,v}_{\mathrm{abt}}(t_1) \stackrel{\mathrm{def}}{=} t_1 \neq t \wedge \mathsf{DSTM.os}(t_1, v)$ indicates that t_1 is being aborted. In other words, any non-aborted thread t is allowed to write to v; if t does not already own location v, then any previous owners are evicted to the abort state.

Validate The $validate(t)$ action has the guard $\mathsf{DSTM.finished}(t)$ and the transition formula given by

$$\forall t_1.\mathsf{DSTM.finished}'(t_1) \equiv$$
$$t_1 \neq t \wedge \mathsf{DSTM.finished}(t_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1)$$
$$\forall t_1.\mathsf{DSTM.validated}'(t_1) \equiv$$
$$t_1 = t \vee \mathsf{DSTM.validated}(t_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1)$$
$$\forall t_1.\mathsf{DSTM.invalid}'(t_1) \equiv$$
$$\mathsf{DSTM.invalid}(t_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1)$$
$$\forall t_1.\mathsf{DSTM.aborted}'(t_1) \equiv$$
$$\mathsf{DSTM.aborted}(t_1) \vee \varphi^{t}_{\mathrm{abt}}(t_1)$$
$$\forall t_1, v_1.\mathsf{DSTM.rs}'(t_1, v_1) \equiv \mathsf{DSTM.rs}(t_1, v_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1)$$
$$\forall t_1, v_1.\mathsf{DSTM.os}'(t_1, v_1) \equiv \mathsf{DSTM.os}(t_1, v_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1)$$

where $\varphi^{t}_{\mathrm{abt}}(t_1) \stackrel{\mathrm{def}}{=} t_1 \neq t \wedge \exists v_1.\mathsf{DSTM.rs}(t, v_1) \wedge \mathsf{DSTM.os}(t_1, v_1)$. In other words, any thread t in the finished state is allowed to validate its read set; in doing so, every owner of a location which t read becomes aborted.

Commit The $commit(t)$ action has $\mathsf{DSTM.validated}(t)$ as the guard, and the transition formula given by the conjunction of

$$\forall t_1.\mathsf{DSTM.finished}'(t_1) \equiv$$
$$t_1 = t \vee \mathsf{DSTM.finished}(t_1) \wedge \neg\varphi^{t}_{\mathrm{inv}}(t_1)$$
$$\forall t_1.\mathsf{DSTM.validated}'(t_1) \equiv$$
$$t_1 \neq t \wedge \mathsf{DSTM.validated}(t_1) \wedge \neg\varphi^{t}_{\mathrm{inv}}(t_1)$$
$$\forall t_1.\mathsf{DSTM.invalid}'(t_1) \equiv$$
$$\mathsf{DSTM.invalid}(t_1) \vee \varphi^{t}_{\mathrm{inv}}(t_1)$$
$$\forall t_1.\mathsf{DSTM.aborted}'(t_1) \equiv$$
$$\mathsf{DSTM.aborted}(t_1) \wedge \neg\varphi^{t}_{\mathrm{inv}}(t_1)$$
$$\forall t_1, v_1.\mathsf{DSTM.rs}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{DSTM.rs}(t_1, v_1)$$
$$\forall t_1, v_1.\mathsf{DSTM.os}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{DSTM.os}(t_1, v_1),$$

where $\varphi^{t}_{\mathrm{inv}}(t_1) \stackrel{\mathrm{def}}{=} t_1 \neq t \wedge \exists v_1.\mathsf{DSTM.os}(t, v_1) \wedge \mathsf{DSTM.rs}(t_1, v_1)$. In other words, any thread t in the validated state is allowed to commit; in doing so, every read by another thread to a location v which t owned becomes invalid, since t is updating v's value.[2]

Abort The $abort(t)$ action for DSTM has the guard $true$ and the transition formula given by the conjunction of

$$\forall t_1.\mathsf{DSTM.finished}'(t_1) \equiv t_1 = t \vee \mathsf{DSTM.finished}(t_1)$$
$$\forall t_1.\mathsf{DSTM.validated}'(t_1) \equiv t_1 \neq t \wedge \mathsf{DSTM.validated}(t_1)$$
$$\forall t_1.\mathsf{DSTM.invalid}'(t_1) \equiv t_1 \neq t \wedge \mathsf{DSTM.invalid}(t_1)$$
$$\forall t_1.\mathsf{DSTM.aborted}'(t_1) \equiv t_1 \neq t \wedge \mathsf{DSTM.aborted}(t_1)$$
$$\forall t_1, v_1.\mathsf{DSTM.rs}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{DSTM.rs}(t_1, v_1)$$
$$\forall t_1, v_1.\mathsf{DSTM.os}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{DSTM.os}(t_1, v_1).$$

The $abort$ action simply clears the state of t.

[2] This is due to the deferred update semantics we've assumed.

C. Model of Transactional Locking II

Transactional locking II is a transaction manager which ensures perceived sequential behavior by giving exclusive locks to locations a thread has written to upon committing, while ensuring that all values read by a transaction have not been altered by others before committing. Our TL2 model uses the status predicates

$$\mathsf{TL2.finished}(t), \mathsf{TL2.validated}(t), \text{ and } \mathsf{TL2.aborted}(t),$$

as well as predicates

$$\begin{aligned} &\mathsf{TL2.rs}(t,v), \mathsf{TL2.ws}(t,v), && \text{read \& write sets} \\ &\mathsf{TL2.ls}(t,v), \mathsf{TL2.ms}(t,v). && \text{lock \& modified sets} \end{aligned}$$

The initial condition

$$\begin{aligned} &\forall t.\mathsf{TL2.status}(t) = \mathsf{finished} \\ &\wedge \forall t,v.\neg\mathsf{TL2.rs}(t,v) \wedge \neg\mathsf{TL2.ws}(t,v) \\ &\quad \wedge \neg\mathsf{TL2.ls}(t,v) \wedge \neg\mathsf{TL2.ms}(t,v) \end{aligned}$$

asserts that threads begin in the finished state, all access sets are initially uninhabited. The system is further constrained by the invariant $\forall t.$ exactly one of

$$\mathsf{TL2.finished}(t), \mathsf{TL2.validated}(t), \mathsf{TL2.aborted}(t)$$

holds. This constraint is an artifact of encoding the three-phase TL2 state with predicates. Although here we list separately TL2's *validate* and *commit* actions, due to the limitation mentioned in Section 6, the model we verify assumes that $validate(t)$ and $commit(t)$ execute together atomically.

Read The $read(t,v)$ action for TL2 has the guard

$$\mathsf{TL2.finished}(t) \wedge (\mathsf{TL2.ms}(t,v) \Rightarrow \mathsf{TL2.ws}(t,v))$$

and the transition formula given by the conjunction of

$$\begin{aligned} &\forall t_1,v_1.\mathsf{TL2.rs}'(t_1,v_1) \equiv \\ &\quad \mathsf{TL2.rs}(t_1,v_1) \vee \neg\mathsf{TL2.ws}(t,v) \wedge t_1 = t \wedge v_1 = v. \end{aligned}$$

In other words, any thread t in the finished state can read v, so long as v is only marked as modified when t has written to v. In that case, v is added to t's read-set if t has not already written to v.

Write The $write(t,v)$ action has the guard $\mathsf{TL2.finished}(t)$ and the transition formula given by the conjunction of

$$\forall t_1,v_1.\mathsf{TL2.ws}'(t_1,v_1) \equiv \mathsf{TL2.ws}(t_1,v_1) \vee t_1 = t \wedge v_1 = v.$$

In other words, any thread t in the finished state can write to v, in which case v is added to t's write-set.

Lock The $lock(t,v)$ action for TL2 has the guard,

$$\mathsf{TL2.finished}(t) \wedge \mathsf{TL2.ws}(t,v),$$

and the transition formula given by the conjunction of

$$\begin{aligned} &\forall t_1.\mathsf{TL2.finished}'(t_1) \equiv \mathsf{TL2.finished}(t_1) \wedge \neg\varphi^{t,v}_{\mathrm{abt}}(t_1) \\ &\forall t_1.\mathsf{TL2.validated}'(t_1) \equiv \mathsf{TL2.validated}(t_1) \wedge \neg\varphi^{t,v}_{\mathrm{abt}}(t_1) \\ &\forall t_1.\mathsf{TL2.aborted}'(t_1) \equiv \mathsf{TL2.aborted}(t_1) \vee \varphi^{t,v}_{\mathrm{abt}}(t_1) \\ &\forall t_1,v_1.\mathsf{TL2.ls}'(t_1,v_1) \equiv \mathsf{TL2.ls}(t_1,v_1) \vee t_1 = t \wedge v_1 = v \end{aligned}$$

where $\varphi^{t,v}_{\mathrm{abt}}(t_1) \stackrel{\mathrm{def}}{=} t_1 \neq t \wedge \mathsf{TL2.ls}(t_1,v)$. In other words, any thread t in the finished state that has written to location v can lock v; by doing so, threads that have previously locked v are evicted to the aborted state.

Validate The $validate(t,v)$ action for TL2 has the guard

$$\begin{aligned} &\mathsf{TL2.finished}(t) \\ &\wedge \neg\exists v_1.\mathsf{TL2.rs}(t,v_1) \wedge \mathsf{TL2.ms}(t,v_1) \\ &\wedge \forall v_1.\mathsf{TL2.ws}(t,v_1) \equiv \mathsf{TL2.ls}(t,v_1) \end{aligned}$$

and the transition formula given by the conjunction of

$$\begin{aligned} &\forall t_1.\mathsf{TL2.finished}'(t_1) \equiv \mathsf{TL2.finished}(t_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1) \\ &\forall t_1.\mathsf{TL2.validated}'(t_1) \equiv \\ &\quad t_1 = t \vee \mathsf{TL2.validated}(t_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1) \\ &\forall t_1.\mathsf{TL2.aborted}'(t_1) \equiv \mathsf{TL2.aborted}(t_1) \vee \varphi^{t}_{\mathrm{abt}}(t_1) \\ &\forall t_1,v_1.\mathsf{TL2.rs}'(t_1,v_1) \equiv \mathsf{TL2.rs}(t_1,v_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1) \\ &\forall t_1,v_1.\mathsf{TL2.ws}'(t_1,v_1) \equiv \mathsf{TL2.ws}(t_1,v_1) \wedge \neg\varphi^{t}_{\mathrm{abt}}(t_1) \end{aligned}$$

where $\varphi^{t}_{\mathrm{abt}}(t_1) \stackrel{\mathrm{def}}{=} t_1 \neq t \wedge \exists v_2.\mathsf{TL2.rs}(t,v_2) \wedge \mathsf{TL2.ws}(t_1,v_2)$. In other words, any thread t in the finished state that has the lock to every location it has written, and has not read from a modified location, is allowed to validate its read set. By doing so, any thread that has written to a location that t has read becomes aborted.

Commit The $commit(t)$ action is guarded by $\mathsf{TL2.validated}(t)$ and has a transition formula given by the conjunction of

$$\begin{aligned} &\forall t_1.\mathsf{TL2.finished}'(t_1) \equiv t_1 = t \vee \mathsf{TL2.finished}(t_1) \\ &\forall t_1.\mathsf{TL2.validated}'(t_1) \equiv t_1 \neq t \wedge \mathsf{TL2.validated}(t_1) \\ &\forall t_1,v_1.\mathsf{TL2.rs}'(t_1,v_1) \equiv t_1 \neq t \wedge \mathsf{TL2.rs}(t_1,v_1) \\ &\forall t_1,v_1.\mathsf{TL2.ws}'(t_1,v_1) \equiv t_1 \neq t \wedge \mathsf{TL2.ws}(t_1,v_1) \\ &\forall t_1,v_1.\mathsf{TL2.ls}'(t_1,v_1) \equiv t_1 \neq t \wedge \mathsf{TL2.ls}(t_1,v_1) \\ &\forall t_1,v_1.\mathsf{TL2.ms}'(t_1,v_1) \equiv t_1 \neq t \\ &\quad \wedge (\mathsf{TL2.ms}(t_1,v_1) \vee \mathsf{TL2.ws}(t,v_1) \wedge \varphi_{\mathrm{act}}(t_1)), \end{aligned}$$

where $\varphi_{\mathrm{act}}(t_1) \stackrel{\mathrm{def}}{=} \exists v_2.\mathsf{TL2.rs}(t_1,v_2) \vee \mathsf{TL2.ws}(t_1,v_2)$. In other words, t can commit after it has validated; by doing so, the locations that t has written to are added to the modified sets of all other active threads.

D. Model of Strict Serializability

The strict serializability system is a deterministic executable specification whose behaviors include exactly the set of strictly serializable traces [16]. Our SS model uses the status predicates $\mathsf{SS.finished}(t)$, $\mathsf{SS.started}(t)$, $\mathsf{SS.invalid}(t)$, and $\mathsf{SS.pending}(t)$, as well as predicates encoding the read-sets $\mathsf{SS.rs}(t,v)$, write-sets $\mathsf{SS.ws}(t,v)$, prohibited read-sets $\mathsf{SS.prs}(t,v)$, prohibited write-sets $\mathsf{SS.pws}(t,v)$, and weak-predecessors $\mathsf{SS.wp}(t,t_1)$ of each thread t. The initial condition

$$\begin{aligned} &\forall t.\mathsf{SS.status}(t) = \mathsf{finished} \wedge \forall t_1,t_2.\neg\mathsf{SS.wp}(t_1,t_2) \\ &\wedge \forall t,v.\neg\mathsf{SS.rs}(t,v) \wedge \neg\mathsf{SS.ws}(t,v) \\ &\quad \wedge \neg\mathsf{SS.prs}(t,v) \wedge \neg\mathsf{SS.pws}(t,v) \end{aligned}$$

asserts that threads begin in the finished state, all access sets are initially uninhabited, and the weak-predecessor relation is empty. SS is further constrained by the invariant, $\forall t.$ exactly one of

$$\mathsf{SS.finished}(t), \mathsf{SS.started}(t), \mathsf{SS.invalid}(t), \mathsf{SS.pending}(t)$$

holds. The constraint is an artifact of encoding the four-phase SS state with predicates.

Read The $read(t,v)$ action for SS has the guard $true$ and the transition formula given by the conjunction of

$$\begin{aligned} &\forall t_1.\mathsf{SS.finished}'(t_1) \equiv \mathsf{SS.finished}(t_1) \wedge \neg\mathsf{SS.ws}(t,v) \Rightarrow t_1 \neq t \\ &\forall t_1.\mathsf{SS.started}'(t_1) \equiv \\ &\quad (\mathsf{SS.ws}(t,v) \Rightarrow \mathsf{SS.started}(t_1)) \\ &\quad \wedge (\neg\mathsf{SS.ws}(t,v) \Rightarrow \\ &\qquad t_1 \neq t \wedge \mathsf{SS.started}(t_1) \\ &\qquad \vee t_1 = t \wedge (\mathsf{SS.started}(t_1) \vee \mathsf{SS.finished}(t_1)) \end{aligned}$$

$$\wedge \neg \mathsf{SS.prs}(t, v))$$
$$\forall t_1.\mathsf{SS.pending}'(t_1) \equiv$$
$$\mathsf{SS.pending}(t_1) \wedge (\neg \mathsf{SS.ws}(t, v) \Rightarrow \neg \varphi^{t,v}_{\mathrm{inv}}(t_1))$$
$$\forall t_1.\mathsf{SS.invalid}'(t_1) \equiv \mathsf{SS.invalid}(t_1) \vee \neg \mathsf{SS.ws}(t, v) \wedge \varphi^{t,v}_{\mathrm{inv}}(t_1)$$
$$\forall t_1, v_1.\mathsf{SS.rs}'(t_1, v_1) \equiv$$
$$\mathsf{SS.rs}(t_1, v_1) \vee \neg \mathsf{SS.ws}(t, v) \wedge t_1 = t \wedge v_1 = v$$
$$\forall t_1, t_2.\mathsf{SS.wp}'(t_1, t_2) \equiv \mathsf{SS.wp}(t_1, t_2)$$
$$\vee \neg \mathsf{SS.ws}(t, v)$$
$$\wedge (t_1 \neq t \wedge t_2 = t \wedge \mathsf{SS.ws}(t_1, v)$$
$$\vee t_1 = t \wedge t_1 \neq t \wedge \mathsf{SS.prs}(t_2, v)$$
$$\vee t_1 = t \wedge \mathsf{SS.pending}(t_2) \wedge \mathsf{SS.finished}(t_1)),$$

where $\varphi^{t,v}_{\mathrm{inv}}(t_1) \stackrel{\mathrm{def}}{=} t_1 = t \wedge \mathsf{SS.prs}(t, v)$. In other words, any thread t can always try to read a location v. If t has already written to v, this action is a no-op. Otherwise, if t has been prohibited from writing to v (i.e., $\mathsf{SS.prs}(t, v)$), then t's status is set to invalid; if not, then t's status is set to started. Since the commit of a write will invalidate the value read from v, t becomes a weak-predecessor of any thread that has written to v,[3] and any thread that has been prohibited from reading v becomes a weak-predecessor of t. Additionally, if this was t's first action for a given transaction, then any pending threads also become weak-predecessors.

Write The $write(t, v)$ action for SS has the guard $true$ and the transition formula given by the conjunction of

$$\forall t_1.\mathsf{SS.finished}'(t_1) \equiv t_1 \neq t \wedge \mathsf{SS.finished}(t_1)$$
$$\forall t_1.\mathsf{SS.started}'(t_1) \equiv$$
$$(\mathsf{SS.started}(t_1) \vee \mathsf{SS.finished}(t_1) \wedge t_1 = t) \wedge \neg \varphi^{t,v}_{\mathrm{inv}}(t_1)$$
$$\forall t_1.\mathsf{SS.pending}'(t_1) \equiv \mathsf{SS.pending}(t_1) \wedge \neg \varphi^{t,v}_{\mathrm{inv}}(t_1)$$
$$\forall t_1.\mathsf{SS.invalid}'(t_1) \equiv \mathsf{SS.invalid}(t_1) \vee \varphi^{t,v}_{\mathrm{inv}}(t_1)$$
$$\forall t_1, v_1.\mathsf{SS.ws}'(t_1, v_1) \equiv \mathsf{SS.ws}(t_1, v_1) \vee t_1 = t \wedge v_1 = v$$
$$\forall t_1, t_2.\mathsf{SS.wp}'(t_1, t_2) \equiv \mathsf{SS.wp}(t_1, t_2)$$
$$\vee t_1 = t \wedge t_2 \neq t \wedge \mathsf{SS.rs}(t_2, v)$$
$$\vee t_1 = t \wedge t_2 \neq t \wedge \mathsf{SS.pws}(t_2, v)$$
$$\vee t_1 = t \wedge \mathsf{SS.pending}(t_2) \wedge \mathsf{SS.finished}(t_1),$$

where $\varphi^{t,v}_{\mathrm{inv}}(t_1) \stackrel{\mathrm{def}}{=} t_1 = t \wedge \mathsf{SS.pws}(t, v)$. In other words, a thread t may always try to write to location v, though if t has been prohibited from writing to v (i.e., $\mathsf{SS.pws}(t, v)$, then t's status is set to invalid; otherwise, t's status is set to started. Any threads that have already read v, or have been prohibited from writing to v, become weak-predecessors of t. Additionally, if this was t's first action for a given transaction, then any pending threads also become weak-predecessors.

Commit The $commit(t)$ action for SS has the guard

$$\neg \mathsf{SS.invalid}(t) \wedge \neg \mathsf{SS.wp}(t, t)$$

and the transition formula given by the conjunction of

$$\forall t_1.\mathsf{SS.finished}'(t_1) \equiv t_1 = t \vee \mathsf{SS.finished}(t_1) \wedge \neg \mathsf{SS.wp}(t, t_1)$$
$$\forall t_1.\mathsf{SS.started}'(t_1) \equiv t_1 \neq t \wedge \mathsf{SS.started}(t_1) \wedge \neg \mathsf{SS.wp}(t, t_1)$$
$$\forall t_1.\mathsf{SS.pending}'(t_1) \equiv t_1 \neq t \wedge \mathsf{SS.pending}(t_1) \vee \varphi^{t}_{\mathrm{pend}}(t_1)$$
$$\forall t_1.\mathsf{SS.invalid}'(t_1) \equiv \mathsf{SS.invalid}(t_1) \vee \varphi^{t}_{\mathrm{inv}}(t_1)$$
$$\forall t_1, v_1.\mathsf{SS.rs}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{SS.rs}(t_1, v_1)$$
$$\forall t_1, v_1.\mathsf{SS.ws}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{SS.ws}(t_1, v_1)$$
$$\forall t_1, v_1.\mathsf{SS.prs}'(t_1, v_1) \equiv t_1 \neq t$$
$$\wedge (\mathsf{SS.prs}(t_1, v_1)$$
$$\vee \mathsf{SS.wp}(t, t_1) \wedge (\mathsf{SS.prs}(t, v_1) \vee \mathsf{SS.ws}(t, v_1)))$$
$$\forall t_1, v_1.\mathsf{SS.pws}'(t_1, v_1) \equiv t_1 \neq t$$
$$\wedge (\mathsf{SS.pws}(t_1, v_1)$$
$$\vee \mathsf{SS.wp}(t, t_1)$$
$$\wedge (\mathsf{SS.pws}(t, v_1) \vee \mathsf{SS.ws}(t, v_1) \vee \mathsf{SS.rs}(t, v_1)))$$
$$\forall t_1, t_2.\mathsf{SS.wp}'(t_1, t_2) \equiv t_1 \neq t \wedge t_2 \neq t$$
$$\wedge (\mathsf{SS.wp}(t_1, t_2)$$
$$\vee \mathsf{SS.wp}(t, t_2)$$
$$\wedge (\mathsf{SS.wp}(t_1, t) \vee \exists v.\mathsf{SS.ws}(t, v) \wedge \mathsf{SS.ws}(t_1, v)))$$

where

$$\varphi^{t}_{\mathrm{inv}}(t_1) \stackrel{\mathrm{def}}{=} t_1 \neq t \wedge \mathsf{SS.wp}(t, t_1)$$
$$\wedge \exists v_1.\mathsf{SS.ws}(t, v_1) \wedge \mathsf{SS.ws}(t_1, v_1), \text{ and}$$
$$\varphi^{t}_{\mathrm{pend}}(t_1) \stackrel{\mathrm{def}}{=} t_1 \neq t \wedge \mathsf{SS.wp}(t, t_1)$$
$$\wedge \neg \exists v_1.\mathsf{SS.ws}(t, v_1) \wedge \mathsf{SS.ws}(t_1, v_1).$$

In other words, any thread t that whose status is not invalid, and is not a self weak-predecessor can commit. In that case, all state involving t is cleared, and the status of every weak-predecessor t_1 of t is either set to invalid, if t_1 had a write-write conflict with t, or pending otherwise. All weak-predecessors of t are then prohibited from reading any location which t has written to, or has itself been prohibited from reading. Similarly, all weak-predecessors of t are prohibited from writing to any location which t has read or written, or has itself been prohibited form writing to. Finally, any weak-predecessor t_2 of t becomes a weak-predecessor of any thread t_1 which t preceded, or had a write-write conflict with.

Abort The $abort(t)$ action for SS has the guard $true$ and the transition formula given by the conjunction of

$$\forall t_1.\mathsf{SS.finished}'(t_1) \equiv t_1 = t \vee \mathsf{SS.finished}(t_1)$$
$$\forall t_1.\mathsf{SS.started}'(t_1) \equiv t_1 \neq t \wedge \mathsf{SS.started}(t_1)$$
$$\forall t_1.\mathsf{SS.pending}'(t_1) \equiv t_1 \neq t \wedge \mathsf{SS.pending}(t_1)$$
$$\forall t_1.\mathsf{SS.invalid}'(t_1) \equiv t_1 \neq t \wedge \mathsf{SS.invalid}(t_1)$$
$$\forall t_1, v_1.\mathsf{SS.rs}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{SS.rs}(t_1, v_1)$$
$$\forall t_1, v_1.\mathsf{SS.ws}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{SS.ws}(t_1, v_1)$$
$$\forall t_1, v_1.\mathsf{SS.prs}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{SS.prs}(t_1, v_1)$$
$$\forall t_1, v_1.\mathsf{SS.pws}'(t_1, v_1) \equiv t_1 \neq t \wedge \mathsf{SS.pws}(t_1, v_1)$$
$$\forall t_1, v_1.\mathsf{SS.wp}'(t_1, t_2) \equiv t_1 \neq t \wedge t_2 \neq t \wedge \mathsf{SS.wp}(t_1, t_2).$$

In other words, a thread t is always abort-enabled, and upon aborting t's state is completely cleared.

[3] Again, the deferred update semantics guides this decision.