# Modularising Opacity Verification for Hybrid Transactional Memory

Alasdair Armstrong and Brijesh Dongol(✉)

Brunel University London, London, UK
{alasdair.armstrong,brijesh.dongol}@brunel.ac.uk

**Abstract.** Transactional memory (TM) manages thread synchronisation to provide an illusion of atomicity for arbitrary blocks of code. There are various implementations of TM, including hardware (HTM) and software (STM). HTMs provide high performance, but are inherently limited by hardware restrictions; STMs avoid these limitations but suffer from unpredictable performance. To solve these problems, hybrid TM (HyTM) algorithms have been introduced which provide reliable software fallback mechanisms for hardware transactions. The key safety property for TM is opacity, however a naive combination of an opaque STM and an opaque HTM does not necessarily result in an opaque HyTM. Therefore, HyTM algorithms must be specially designed to satisfy opacity. In this paper we introduce a modular method for verifying opacity of HyTM implementations. Our method provides conditions under which opacity proofs of HTM and STM components can be combined into a single proof of an overall hybrid algorithm. The proof method has been fully mechanised in Isabelle, and used to verify a novel hybrid version of a transactional mutex lock.

## 1 Introduction

By allowing programmers to mark blocks of arbitrary code as *transactions*, Transactional Memory (TM) aims to provide an easy-to-use synchronisation mechanism for concurrent access to shared data. Unlike coarse-grained locking, TM implementations are fine-grained, which improves performance. In recent years, TM has appeared as software libraries in languages such as Java, Clojure, Haskell and C++11, and received hardware support in processors (e.g., Intel's TSX).

*Software Transactional Memory* (STM), as provided by the aforementioned software libraries, offers a programmer-friendly mechanism for shared-variable concurrency. However, it suffers from unpredictable performance which makes it unsuitable for some applications. On the other hand, *Hardware Transactional Memory* (HTM), as implemented in modern Intel processors, offers high performance but comes with many limitations imposed by the constraints of the hardware itself. For example, HTM implementations do not guarantee progress for a transaction even in the absence of other concurrent transactions [11]. *Hybrid*

© IFIP International Federation for Information Processing 2017
Published by Springer International Publishing AG 2017. All Rights Reserved
A. Bouajjani and A. Silva (Eds.): FORTE 2017, LNCS 10321, pp. 33–49, 2017.
DOI: 10.1007/978-3-319-60225-7_3

*TM (HyTM)* implementations address these issues by integrating STM and HTM [10]. Recent work [2,3,18] has focused on providing software fallbacks for HTM, combining the performance benefits of HTM with the strong semantics and progress guarantees of STM.

*Opacity* [8,9] is the primary safety property for TM, which ensures that implementations have the familiar properties of database transactions: atomicity, consistency, and isolation. Opacity requires that all transactions (including aborting ones) can be serialised into some meaningful sequential order, so that no transaction witnesses an inconsistent state caused by the partial execution of any other transaction. Overall, this ensures that TM implementations execute with an *illusion of atomicity.*

HyTM algorithms, which are the focus of this paper, consist of STM (*slow-path*) transactions executing in parallel with HTM (*fast-path*) transactions. Since an execution may consist of only STM or only HTM transactions, one must ensure that slow-path and fast-path transactions are by themselves opaque. In addition, synchronisation between slow- and fast-path transactions must be introduced to ensure that executions consisting of arbitrary combinations of these transactions is opaque. It is already known that naively combining STM and HTM results in a non-opaque HyTM algorithm [2]. In this paper, we develop a modular verification method for proving opacity of HyTM algorithms—our method provides a means for independently proving opacity of both the STM slow path and the HTM fast path, and then combining them into a proof of opacity of the overall system.

To demonstrate our proof method, in Sect. 2, we develop a novel hybrid version of Dalessandro *et al.*'s Transactional Mutex Lock [4], extending it with a subscription mechanism described in [3]. Our algorithm, HyTML, combines an *eager* STM, where writes to the shared store are immediately committed to memory, with a *lazy* fast path HTM, where writes to the shared store are cached until the HTM executes a commit operation. Moreover, it improves concurrency in the original TML algorithm by allowing multiple concurrent writing HTM transactions; in the original algorithm, all transactions abort in the presence of *any* concurrent writing transaction.

Our proof method is an extension of previous work [6,14] that uses trace refinement of I/O automata (IOA) [17] to verify opacity via a TM specification known as TMS2 [7]. Unlike existing work, our methods enable one to verify HyTML in a modular manner (i.e., by combining *individual opacity proofs* of the fast-path and slow-path components) despite the monolithic structure of the algorithm. Our proof methods are influenced by compositional techniques [20] such as rely/guarantee [12]. However, unlike rely-guarantee, which focusses on composing processes, we focus on composition at the level of *components*, which themselves consist of multiple parallel processes.

We start by developing the notion of an *interference automaton* (Sect. 4), which specialises IOA by including transitions that take into account any potential interference from the environment. Parallel composition for interference automata is developed in Sect. 5, and the notion of *weak simulation* for parallel interference automata is given in Sect. 6. There we provide our main decomposition theorem, which describes how weak simulations can be combined to ensure trace refinement of the composed system. We apply our proof methods to verify HyTML in Sect. 7; we show how individual opacity proofs for the STM and HTM components can be combined, to avoid the complexity inherent in a monolithic proof. All the proofs in this paper, including our meta-theory, have been mechanised[1] in the Isabelle theorem prover [19].

## 2 Hybrid TML

Our running example is the *Hybrid Transaction Mutex Lock* (*HyTML*) algorithm given in Listing 1, which extends Dalessandro *et al.*'s TML algorithm [4] with a 2-counter subscription mechanism [3]. HyTML synchronises the software *slow path* with a hardware *fast path* using *glb* (which is *published* by software and *subscribed* by hardware) and *ctr* (which is *published* by hardware and *subscribed* by software).

The parity of *glb* indicates whether a writing software transaction is currently executing. Namely, a writing software transaction increments *glb* once at Line 31, where it effectively acquires the write lock, and again at Line 37, where it effectively releases the write lock. Thus, $\lfloor glb/2 \rfloor$ gives the total number of committed software transactions. TML, and by extension HyTML, has the property

**Listing 1.** A Hybrid Transactional Mutex Lock (HyTML) algorithm

```
 1: procedure INIT
 2:     $glb, ctr \leftarrow 0, 0$

 3: procedure FPBegin$_t$
 4:     XBegin()
 5:     $loc_t \leftarrow glb$
 6:     $writer_t \leftarrow false$
 7:     if $odd(loc_t)$ then
 8:         XAbort()

 9: procedure FPRead$_t(a)$
10:     return $*a$

11: procedure FPWrite$_t(a, v)$
12:     $writer_t \leftarrow true$
13:     $*a \leftarrow v$
14: procedure FPCommit$_t$
15:     if $writer_t$ then
16:         $ctr{+}{+}$
17:     XEnd()

18: procedure SPBegin$_t$
19:     repeat
20:         $loc_t \leftarrow glb$
21:         $lctr_t \leftarrow ctr$
22:     until $even(loc_t)$

23: procedure SPRead$_t(a)$
24:     $v_t \leftarrow *a$
25:     if $glb = loc_t$ then
26:         if $ctr = lctr_t$ then
27:             return $v_t$
28:     abort

29: procedure SPWrite$_t(a, v)$
30:     if $even(loc_t)$ then
31:         if $\neg dcss(\&glb, loc_t, \&ctr, lctr_t, loc_t{+}1)$
32:         then abort
33:         else $loc_t{+}{+}$
34:     $*a \leftarrow v$

35: procedure SPCommit$_t$
36:     if $odd(loc_t)$ then
37:         $glb \leftarrow loc_t + 1$
```

[1] The Isabelle files may be downloaded from [1].

that only a single software transaction can be writing at a time. The presence of a software writer causes all concurrently executing transactions, including fast-path transactions, to abort.[2] Unlike TML, HyTML allows more than one concurrent writing transaction via the fast path. Variable $ctr$ is used to signal a completed hardware transaction and is incremented whenever a writing hardware transaction commits (Line 16). The total number of committed writing transactions is therefore given by $\lfloor glb/2 \rfloor + ctr$. Note that read-only transactions modify neither $glb$ nor $ctr$.

*Software Slow Path.* The software slow path implementation is a conservative extension to the original TML algorithm [4] — we refer the interested reader to [4,5] for further details of the behaviour of TML. The implementation consists of operations `SPBegin` and `SPCommit` that start and end software transactions, respectively, as well as `SPRead` and `SPWrite` that perform (software) transactional reads and writes, respectively. Each operation and transaction-local variable is indexed by a transaction identifier $t$.

Procedure $\texttt{SPBegin}_t$ repeatedly polls both $glb$ and $ctr$, storing their values in local variables $loc_t$ and $lctr_t$, respectively, until $loc_t$ is even. This ensures that a software transaction cannot begin until there are no software writers. Procedure $\texttt{SPRead}_t(a)$ first reads the value in address $a$ from memory and then checks (non-atomically) if $glb$ and $ctr$ are consistent with $loc_t$ and $lctr_t$, respectively. The value of the address is returned if both checks succeed, otherwise it is likely that the transaction $t$ has witnessed an inconsistent snapshot of the store, and hence it aborts.

Procedure $\texttt{SPWrite}_t$ first checks the parity of $loc_t$. If $loc_t$ is odd, then $t$ must itself be the (unique) software writer, i.e., $t$ had acquired the mutex lock from a previous call to $\texttt{SPWrite}_t$. Therefore, $t$ can immediately proceed and eagerly update the value of $*a$ in the store to $v$. If $loc_t$ is even, it contends with other writers to acquire the lock using a *double compare single swap* operation: *dccs*, which atomically checks both $lctr_t$ and $loc_t$ against their global values and updates $glb$ to $loc_t + 1$ if both are unmodified (which effectively acquires the mutex lock). The *dccs* operation returns *true* iff it is successful. If either $glb$ or $ctr$ have changed since $t$ first read their values within $\texttt{SPBegin}_t$, then $t$ may go on to construct an inconsistent memory state, and hence, it must abort. Otherwise (i.e., if *dccs* succeeds), $loc_t$ is incremented (Line 33) to match the new value of $glb$. This makes the value of $loc_t$ odd, allowing the expensive *dccs* operation to be elided in future calls to $\texttt{SPWrite}_t$, as explained above, and allows future calls to $\texttt{SPRead}_t$ to succeed.

Procedure $\texttt{SPCommit}_t$ always succeeds. It checks to see if $t$ is a writing transaction (i.e., $loc_t$ is odd). If so, $loc_t$ must be equal to $glb$, and hence, the update to $glb$ at Line 37 is equivalent to an increment of $glb$ that makes $glb$'s value even. This second increment effectively releases the mutex lock.

[2] There are some exceptions, e.g., a read-only software transaction can successfully commit even in the presence of another writer if no more reads are performed [5].

*Hardware Fast Path.* Our implementation uses HTM primitives provided by an Intel x86 processor with TSX extensions. However, we keep the specifics of the hardware generic and assume as little as possible about the behaviour of the primitives, allowing our work to more easily be adapted to work with other HTMs. We use three basic primitives: `XBegin`, which starts a hardware transaction, `XEnd`, which ends (and attempts to commit) the hardware transaction, and `XAbort`, which explicitly aborts the hardware transaction. We assume that, once started, a hardware transaction may be forced abort at any time for any reason, which is consistent with Intel's specifications [11]. In addition, when interference on any variable that has been read is detected, a fast-path transaction *must* abort (details are provided below).

Procedure $\texttt{FPBegin}_t$ starts a fast-path transaction by calling `XBegin`, then *subscribes* to the software global version number, $glb$, by reading and recording its value in a local variable $loc_t$. A local flag $writer_t$ (initially $false$) is used to indicate whether a fast-path transaction is a writer. Transaction $t$ only begins if $loc_t$ is even—if $loc_t$ is odd, a slow-path writer is executing, and hence, the fast-path transaction aborts.

Note that because the read of $glb$ occurs after `XBegin`, the underlying HTM will track the value of $glb$ in memory, ensuring that the fast-path transaction aborts if $glb$ changes. Such checks to $glb$ are performed automatically by the HTM outside the control of the fast-path implementation, and hence, is not explicit in the code (Listing 1). This behaviour is captured in our model of the fast-path transactions by validating that the value of $glb$ is equal to $loc_t$ for every step of fast-path transaction $t$, and aborting whenever this validation fails.

The fast-path read and write operations, $\texttt{FPRead}_t$ and $\texttt{FPWrite}_t$ consist of standard memory operations, but the underlying HTM will ensures these writes are not visible outside $t$ until $t$ commits. In $\texttt{FPWrite}_t$, the flag $writer_t$ is set to $true$ to indicate that $t$ is now a writer. Procedure $\texttt{FPCommit}_t$ updates $ctr$ if $t$ is a writer, which indicates to software transactions that a fast-path transaction is committing. Note that this increment to $ctr$ will not cause other fast-path transactions to abort. Finally, $\texttt{FPCommit}_t$ calls `XEnd`, which, for a writer transaction, commits all the pending writes to the store and *publishes* the increment to $ctr$.

## 3 The TMS2 Specification

The basic principle behind the definition of opacity (and other similar definitions) compares a given concurrent history of transactional operations against a sequential one. Opacity requires it be possible for transactions to be serialised so that the real-time order of transactions is preserved. Within this serialisation order, read operations for all transactions, including aborted transactions, must be consistent with the state of the memory store, which is obtained from the initial store by applying the previously committed transactions in their serialised order [8,9]. We elide the formal definition of opacity here, focusing instead on an automata-based TM specification, TMS2 [7]. Lesani et al. [15] have mechanically verified that TMS2 is opaque, thus it is sufficient to show trace refinement against TMS2 to verify opacity of an implementation (cf [6,14]). TMS2 and the implementations we verify are modelled using input/output automata [16,17].

**Definition 1.** *An* I/O automaton (IOA) *is a labelled transition system $A$ with a set of states $states(A)$, a set of actions $acts(A)$ (partitioned into internal and external actions), a set of start states $start(A) \subseteq states(A)$ and a transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ (so that the actions label the transitions).*

TMS2 contains external transitions modelling operation invocations and responses, e.g., the invoke and respond actions for a write action are given below, where $status_t$ is a transaction-local variable that models control flow. The transition is *enabled* if all its preconditions, given after the keyword Pre, hold in the current state. State modifications (effects) of a transition are given as assignments after the keyword Eff.

```
$inv_t(\texttt{TMWrite}(a, v))$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doWrite}(a, v)$
```

```
$resp_t(\texttt{TMWrite})$
Pre: $status_t = \text{writeResp}$
Eff: $status_t := \text{ready}$
```

TMS2 contains a pair of invocations and responses for begin, read, write and commit operations. In addition, a response is provided for aborting operations:

```
$resp_t(\texttt{TMAbort})$
Pre: $status_t \notin \{\text{notStarted}, \text{ready}, \text{commitResp}, \text{committed}, \text{aborted}\}$
Eff: $status_t := \text{aborted}$
```

After invoking a write, read, or commit operation, a transaction may execute one of the 'do' actions in Fig. 1, which performs the corresponding abstract operation.

```
$\texttt{DoRead}_t(a, n)$
Pre: $status_t = \text{doRead}(a)$
     $a \in dom(wrSet_t) \vee validIdx_t(n)$
Eff: if $a \in dom(wrSet_t)$ then
       $status_t := \text{readResp}(wrSet_t(a))$
     else $v := memSeq(n)(a)$
       $status_t := \text{readResp}(v)$
       $rdSet_t := rdSet_t \oplus \{a \mapsto v\}$
```

```
$\texttt{DoWrite}_t(a, v)$
Pre: $status_t = \text{doWrite}(a, v)$
Eff: $status_t := \text{writeResp}$
     $wrSet_t := wrSet_t \oplus \{a \mapsto v\}$
```

```
$\texttt{DoCommitRO}_t(n)$
Pre: $status_t = \text{doCommit}$
     $dom(wrSet_t) = \emptyset$
     $validIdx_t(n)$
Eff: $status_t := \text{commitResp}$
```

```
$\texttt{DoCommitW}_t$
Pre: $status_t = \text{doCommit}$
     $rdSet_t \subseteq latestMem$
Eff: $status_t := \text{commitResp}$
     $memSeq := memSeq \oplus newMem_t$
```

**where**
$$\begin{aligned} maxIdx &\mathrel{\widehat{=}} max(dom(memSeq)) \\ latestMem &\mathrel{\widehat{=}} memSeq(maxIdx) \\ newMem_t &\mathrel{\widehat{=}} \{maxIdx + 1 \mapsto (latestMem \oplus wrSet_t)\} \\ validIdx_t(n) &\mathrel{\widehat{=}} beginIdx_t \leq n \leq maxIdx \wedge rdSet_t \subseteq memSeq(n) \end{aligned}$$

**Fig. 1.** Internal actions of TMS2

TMS2 guarantees that transactions satisfy two critical requirements: **(R1)** all reads and writes of a transaction work with a *single consistent memory snapshot* that is the result of all previously committed transactions, and **(R2)** the *real-time order* of transactions is preserved. Full details of TMS2 may be found in [7]. Here, we give a brief overview of the requirements that our implementation must satisfy.

To ensure **(R1)**, the state of TMS2 includes $\langle memSeq(0), \ldots\ memSeq\ (maxIdx)\rangle$, which is a sequence of all possible memory snapshots (the *stores sequence*). Initially the sequence consists of one element, the initial memory $memSeq(0)$. Committing writer transactions append a new memory $newMem$ to this sequence (cf. $\texttt{DoCommitW}_t$), by applying the writes of the transaction to the last element $memSeq(maxIdx)$. To ensure that the writes of a transaction are not visible to other transactions before committing, TMS2 uses a *deferred update* semantics: writes are stored locally in the transaction $t$'s write set $wrSet_t$ and only published to the shared state when the transaction commits. Note that this does not preclude TM implementations with eager writes, such as TML. However eager implementations must guarantee that writes are not observable until after the writing transaction has committed.

Each transaction $t$ keeps track of all its reads from memory in a read set $rdSet_t$. A read of address $a$ by transaction $t$ checks that either $a$ was previously written by $t$ itself (**then** branch of $\texttt{DoRead}_t(a)$), or that all values read so far, including $a$, are from the same memory snapshot $n$, where $beginIdx_t \leq n \leq maxIdx$ (predicate $validIdx_t(n)$ from the precondition, which must hold in the **else** branch). In the former case the value of $a$ from $wrSet_t$ is returned, and in the latter, the value from $memSeq(n)$ is returned and the read set is updated. The read set of $t$ is also validated when a transaction commits (cf. $\texttt{DoCommitRO}_t$ and $\texttt{DoCommitW}_t$). Note that when committing, a read-only transaction may read from a memory snapshot older than $memSeq(maxIdx)$, but a writing transaction must ensure that all reads in its read set are from most recent memory (i.e. $latestMem\ memSeq(maxIdx)$), since its writes will update the memory sequence with a new snapshot.

To ensure **(R2)**, if a transaction $t'$ commits before transaction $t$ starts, then the memory that $t$ reads from must include the writes of $t'$. Thus, when starting a transaction, $t$ saves the current last index of the memory sequence, $maxIdx$, into a local variable $beginIdx_t$. When $t$ performs a read, the predicate $validIdx_t(n)$ ensures that that the snapshot $memSeq(n)$ used has $beginIdx_t \leq n$, implying that writes of $t'$ are included.

Our proof of opacity is based on *trace refinement* [16] between HyTML and TMS2, which ensures that every externally visible execution of HyTML is a possible externally visible execution of TMS2. Since every execution of TMS2 is known to be opaque [15], one can conclude that HyTML is itself opaque. We develop a proof method for trace refinement that exploits the fact that HyTML consists of two distinct sets of transactions: slow- and fast-path. Namely, our method proves opacity of each set of transactions independently, taking into account any possible interference from the other set.

## 4 Interference Automata

In this section, we formalise the concept of *interference automata* and the notions of trace refinement and forward simulation that we use. Interference automata specialise IOA by explicitly including transitions for *environment steps*, representing the potential interference from other components within the same system. In the context of the HyTM implementations we verify, an interference automaton will model the fast-path (slow-path) transactions with interference stemming from the slow-path (fast-path).

**Definition 2 (Interference automata).** *An* interference automaton $A$ *consists of:*

- $P_A$ *is an (infinite) set of process identifiers,*
- *sets* $local(A)$ *and* $global(A)$ *of local and global states,*
- *sets* $external(A)$ *and* $internal(A)$ *of external and internal actions, and*
- *an environment action* $\epsilon \notin external(A) \cup internal(A)$.

*We assume* $external(A) \cap internal(A) = \emptyset$, *and use* $actions(A) = external(A) \cup internal(A) \cup \{\epsilon\}$ *to denote the actions of* $A$. *Furthermore:*

- initialisation *of* $A$ *is described by*
  - $lstart(A) \subseteq P_A \to local(A)$, *a set of local start states, and*
  - $gstart(A) \in global(A)$, *a global start state*
- transitions *of* $A$ *are described by*
  - $ltrans(A) \subseteq (local(A) \times global(A)) \times actions(A) \times (local(A) \times global(A))$, *which describes local transitions, and*
  - $env(A) \subseteq global(A) \times global(A)$, *which is a reflexive relation that describes environment transitions.*

The overall state space of $A$ is given by $states(A) = (P_A \to local(A)) \times global(A)$. That is, a state is a pair consisting of a local state for every possible process in $P_A$ and a global state. For any state $s$, the local part of the state is denoted by $s_l$, and the global part by $s_g$, and hence, $s = (s_l, s_g)$.

An interference automaton $A$ may perform an environment transition in $env(A)$, which may only modify the global state, or a local transition for a specific process $p \in P_A$, which may only modify the local state of $p$ and the global state. For states $s$ and $s'$, action $a$, and process $p$, we denote an internal or external transition of $A$ by $s \xrightarrow{a,p}_A s'$, where the action is paired with the process identifier executing the action. By construction, we have that the local state of process $p'$ is unchanged after a transition of process $p$ whenever $p \neq p'$. For global state $s_g, s'_g$, we use $s_g \xrightarrow{\epsilon}_A s'_g$ to denote an environment transition, which is lifted to the level of states in the obvious way. Namely, if $s_l$ is a local state, we let $(s_l, s_g) \xrightarrow{\epsilon}_A (s_l, s'_g)$ denote an environment transition.

A *run* of an interference automaton $A$ is an alternating sequence of states and actions starting from an initial state. The *traces* of $A$, denoted $traces(A)$, are the runs of $A$ restricted to external actions, and the *reachable states* of $A$, denoted $reach(A)$, are states that can be reached by some run of $A$. For interference automata $A$ and $C$, we say $C$ is a *trace refinement* of $A$ iff $traces(C) \subseteq traces(A)$.

Interference automata may be regarded as a special case of IOA, where the state is specialised and actions are split into internal and environment actions. Therefore, all theorems of IOA, including notions of simulation [16] are also applicable in this setting. Note that an interference automaton $A$ represents the actions of an arbitrary amount of processes, which is why $P_A$ must be infinite. As such, interference automata represent *systems of processes* and not specific sets of processes. A *forward simulation* is a standard way of verifying trace refinement between a concrete implementation and an abstract specification. For interference automata, this involves proving simulation between the external, internal, and environment steps.

**Definition 3 (Forward simulation).** *If $A$ and $C$ are interference automata such that $external(C) \subseteq external(A)$, we say $R \subseteq states(C) \times states(A)$ is a* forward simulation *between $A$ and $C$ iff each of the following hold:*

Initialisation. $\forall cs \in start(C) \bullet \exists as \in start(A) \bullet (cs, as) \in R$

External step correspondence

$$\begin{aligned}&\forall cs \in reach(C),\ as \in reach(A),\ a \in external(C),\ p \in P_C, cs' \in states(C) \bullet\\ &\quad (cs, as) \in R \wedge cs \xrightarrow{a,p}_C cs' \implies\\ &\qquad \exists as' \in states(A) \bullet (cs', as') \in R \wedge as \xrightarrow{a,p}_A as',\end{aligned}$$

Internal step correspondence

$$\begin{aligned}&\forall cs \in reach(C), as \in reach(A), a \in internal(C), p \in P_C, cs' \in states(C) \bullet\\ &\quad (cs, as) \in R \wedge cs \xrightarrow{a,p}_C cs' \implies (cs', as) \in R \vee\\ &\qquad \exists as' \in states(A), a' \in internal(A) \bullet (cs', as') \in R \wedge as \xrightarrow{a',p}_A as',\end{aligned}$$

Environment step correspondence

$$\begin{aligned}&\forall cs \in reach(C), as \in reach(A), cs'_g \in global(C) \bullet\\ &\quad (cs, as) \in R \wedge cs_g \xrightarrow{\epsilon}_C cs'_g \implies\\ &\qquad \exists as'_g \in global(A) \bullet ((cs_l, cs'_g), (as_l, as'_g)) \in R \wedge as_g \xrightarrow{\epsilon}_A as'_g.\end{aligned}$$

Soundness of the forward simulation rule with respect to trace refinement has been checked in Isabelle [1].

**Theorem 1 (Soundness).** *If $R$ is a forward simulation between interference automata $A$ and $C$, then $C$ is a trace refinement of $A$, i.e., $traces(C) \subseteq traces(A)$.*

In Sect. 5, we introduce the concept of parallel interference automata and in Sect. 6, we develop a theorem for decomposing parallel interference automata into proofs of individual sub-components. It turns out that our decomposition theorem only needs assume the existence of *weak forward simulation* of the components, in which environment step correspondence may not hold. The notion of a weak simulation is important here, as weak simulations correspond to our existing proofs of opacity for e.g. TML, since these proofs do not involve environment steps. As such, this facilitates the re-use of existing proofs of STM components in the parallel case with only minor modifications. Note that weak simulation between $A$ and $C$ ensures trace refinement for any automaton $C$ in which $env(C)$ is the identity relation since the environment step correspondence proof is trivial.

## 5 Parallel Interference Automata

In this section, we define a notion of parallel composition for interference automata. The idea is that any possible interference from one component of the parallel composition is reflected as an environment transition in the other. Thus, the parallel composition $B \| C$ comprises an interleaving of the local (internal and external) actions of both $B$ and $C$.

Two interference automata $B$ and $C$ can be composed iff they are *compatible*, which only requires that they share the same start state, i.e., $gstart(B) = gstart(C)$. We let $\uplus$ denote disjoint union with injections (or inclusion maps) $\iota_1$ and $\iota_2$.

**Definition 4 (Parallel composition).** *The* parallel composition *of two compatible interference automata $B$ and $C$ is constructed as follows:*

- $local(B\|C) = local(B) \uplus local(C)$,
- $global(B\|C) = global(B) \cup global(C)$,
- $P_{B\|C} = P_B \uplus P_C$,
- $lstart(B\|C) = \{f \cup g \bullet f \in lstart(B) \wedge g \in lstart(C)\}$,
- $gstart(B\|C) = gstart(B) = gstart(C)$ *as $B$ and $C$ are compatible,*
- $internal(B\|C) = internal(B) \uplus internal(C)$,
- $((\iota_n(s), g), \iota_n(a), (\iota_n(s'), g')) \in ltrans(B\|C)$ *iff* $((s, g), a, (s', g')) \in ltrans(B)$ *when* $n = 1$ *and* $((s, g), a, (s', g')) \in ltrans(C)$ *when* $n = 2$, *and*
- $env(A) = Id$, *where $Id$ is the identity relation.*

Essentially this construction splits both the processes and the internal state space of the automaton into left and right processes and states, respectively. An invariant of any composed automaton is that left processes always act on left internal states, and vice versa. For the parallel composition $B\|C$, we typically refer to the automaton $B$ as the left automaton and $C$ as the right automaton. We use $\mathcal{L}$ to denote the projection function that takes a combined state of $B\|C$ and projects just to the part from the left automata $B$, and similarly for $\mathcal{R}$ and $C$.

Henceforth, we make the environment transitions of interference automata explicit. We introduce the notation $I \rhd A$ for an interference automaton $A$ where the environment is the relation $I$, i.e. $env(I \rhd A) = I$. We write $A$ when $env(A) = Id$ and refer to such $A$ as an *interference-free automaton*. Note that we therefore have $Id \rhd A = A$.

In Definition 4, the environment of the composed interference automaton $(I_C \rhd B)\|(I_B \rhd C)$ is set to be the identity relation $Id$, which is possible under the assumption that the local transitions of $I_C \rhd B$ imply the environment transitions of $C$ (namely $I_B$), and vice versa. To use this assumption in our proofs, we introduce the notion of a *guarantee condition* (inspired by rely/guarantee reasoning [12]). We say that an automaton $I \rhd B$ *guarantees* a relation $J$ when

$$\forall s \in \mathit{reach}(I \rhd B), a \in \mathit{actions}(I \rhd B), p \in P_B \bullet s \xrightarrow{a,p}_{I \rhd B} s' \implies (s_g, s'_g) \in J.$$

This states that every reachable transition in $I \rhd B$ modifies the global state only as permitted by $J$. In other words, if $I_C \rhd B$ guarantees $I_B$ and $I_B \rhd C$ guarantees $I_C$, then this ensures that every local transition of $I_C \rhd B$ can be matched with a environment step of $I_B \rhd C$, and vice versa.

As mentioned an (interference-free) interference automaton $A$ represents the actions of zero or more transactions of type $A$. Similarly the parallel composition $A \| A$ also represents zero or more transactions of type $A$, with some labelled as from the left $A$ and others from the right. In other words, parallel composition is idempotent for interference free interference automata. This can be shown via a re-labelling of process identifiers, and has been verified in Isabelle (see [1]).

**Theorem 2.** *$\mathit{traces}(A \| A) = \mathit{traces}(A)$.*

We will use this theorem in the proof of HyTML to split the interference-free IOA specification TMS2 into the parallel composition of two TMS2 components. Thus, to show that HyTML is a trace refinement of TMS2, it will be sufficient to show that the software and hardware components individually are refinements of TMS2.

## 6 Simulation Proofs for Parallel Interference Automata

For modular verification of a parallel interference automaton, we provide a way to build a simulation of a parallel composition from individual weak simulations of the sub-components. For example, in HyTML we consider the two concrete fast/slow paths, and prove both of them TMS2 independently. By Theorem 2, we have that $\mathit{traces}(\text{TMS2}) = \mathit{traces}(\text{TMS2} \| \text{TMS2})$, and hence, for modular proofs of opacity, it is sufficient to consider abstract specifications of the form $A \| A$.

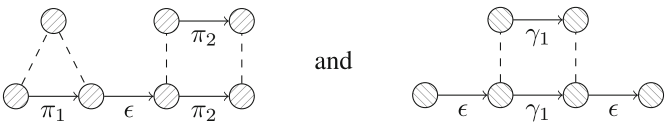
Consider interference automata $I_C \rhd B$ and $I_B \rhd C$, and an abstract interference automaton $I_A \rhd A$. Assume we have weak simulations $R$ and $S$ where

$$I_A \rhd A \text{ weakly simulates } I_C \rhd B \qquad \text{and,} \qquad I_A \rhd A \text{ weakly simulates } I_B \rhd C.$$

We aim to develop conditions such that $R \| S$ is a full (non-weak) simulation between $A \| A$ and $B \| C$, where

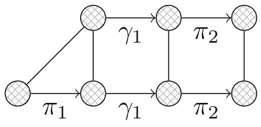
$$R \| S = \{(s, s') \bullet (\mathcal{L}(s), \mathcal{L}(s')) \in R \wedge (\mathcal{R}(s), \mathcal{R}(s')) \in S\}.$$

We now describe the weak simulations $R$ and $S$, including the state projection functions $\mathcal{L}$ and $\mathcal{R}$, and their interaction with the non-weak simulation of the whole system. Graphically, we can visualise weak simulations $R$ and $S$ as

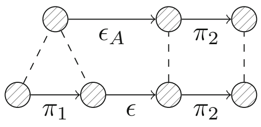


where the local states of the left (right) automaton $I_C \triangleright B$ (symmetrically $I_B \triangleright C$) combined with the global state is represented by ⊘ ( ⦸ ). Thus, the left (right) simulation $R$ ($S$) is over ⊘ ( ⦸ ). Each state of the parallel automaton $B \| C$, denoted ⊗, contains both left and right processes, their local states, as well as the shared global state.

For the weak simulations $R$ and $S$, we must construct a simulation $R \| S$ of the form:



where the environment step $\epsilon$ of $R$ must correspond to the appropriate program step of $S$, namely $\gamma_1$. However, we cannot prove this without some additional properties, because we do not know how actions of $I_B \triangleright C$ affect $R$, and similarly for $I_C \triangleright B$ and $S$. Note that establishing environment step correspondence (which would turn $R$ and $S$ into non-weak forward simulations) would not help. For example, consider $R'$:



Because we have no way of guaranteeing that the abstract state after $\epsilon_A$ in $R'$ is the same as the abstract state after $\gamma_1$ in $S$, we cannot naively construct a parallel forward simulation. Instead we use *non-interference* conditions which guarantee that $C$ and $B$ do not affect $R$ and $S$, respectively. In essence, this enables us to 'stitch' together the two simulations $R$ and $S$ into a simulation of the parallel composition. In other words, the simulation relations used in both component proofs are preserved by the effects of both components' actions on the global state.

**Definition 5.** *The condition nonInterferenceLeft*$(R, S, B, C, A)$ *holds iff*

$$\begin{aligned}
&\forall c_l,\ c_g,\ a_l,\ a_g,\ \pi_C,\ \pi_A,\ p \bullet\\
&\ \mathcal{L}(c_l, c_g) \in \mathit{reach}(B) \wedge \mathcal{L}(a_l, a_g) \in \mathit{reach}(A) \wedge \mathcal{R}(c_l, c_g) \in \mathit{reach}(C)\\
&\ \ \wedge (\mathcal{R}(c_l, c_g), \mathcal{R}(a_l, a_g)) \in R \wedge (\mathcal{L}(c_l, c_g), \mathcal{L}(a_l, a_g)) \in S\\
&\ \ \wedge \mathcal{L}(c_l, c_g) \xrightarrow{\pi_B, p}_B \mathcal{L}(c_l', c_g') \wedge \mathcal{L}(a_l, a_g) \xrightarrow{\pi_A, p}_A \mathcal{L}(a_l', a_g')\\
&\ \Longrightarrow\ (\mathcal{R}(c_l, c_g'), \mathcal{R}(a_l, a_g')) \in R.
\end{aligned}$$

*where* $\pi_A$ *and* $\pi_C$ *are corresponding actions. Symmetrically, we define a condition nonInterferenceRight*$(R, S, B, C, A)$.

The reason these conditions are needed is that our guarantee conditions talk purely about the state changes caused by the automaton itself, but not about the simulation relations between automata. While these non-interference conditions at first look complicated due to the amount of notation involved, notice that the local state $c_l$ and $a_l$ does not change between the pre- and post-condition for the simulation relation $R$. What this means is that we are really showing only that effects contained within the guarantee conditions do not affect the simulation relation, which means that these conditions turn out to be quite straightforward to prove in practice, as will be seen in Sect. 7.

Attempting to remove these non-interference conditions to make the method fully compositional might not be worthwhile in practice, as doing so would require full (rather than weak) simulations for each of the components. This proves to be difficult, as it requires induction on the amount of interference within the simulation proof of each component, and it would preclude easy re-use of existing opacity proofs for the fast and slow paths.

We can now state our simulation theorem for parallel interference automata. The theorem states that $R \| S$ can be strengthened to a forward simulation between $B \| C$ and $A \| A$ provided $R$ ($S$) is a weak simulation between $B$ ($C$) and $A$, and certain guarantee and non-interference conditions hold. This theorem has been verified in Isabelle [1].

**Theorem 3 (Decomposition).** *For two compatible interference automata $B$ and $C$, if $R$ is a weak forward simulation between $I_A \rhd A$ and $I_C \rhd B$, and $S$ is a weak forward simulation between $I_A \rhd A$ and $I_B \rhd C$, where*

- *$I_B \rhd C$ guarantees $I_c$, and $I_C \rhd B$ guarantees $I_B$,*
- *$nonInterferenceRight(R, S, I_C \rhd B, I_B \rhd C, I_A \rhd A)$,*
- *$nonInterferenceLeft(R, S, I_C \rhd B, I_B \rhd C, I_A \rhd A)$.*

*Then $R \| S$ is a (non-weak) forward simulation between $B \| C$ and $A \| A$, and hence $traces(B \| C) \subseteq traces(A \| A)$.*

## 7 HyTML Proof and Mechanisation

In this section we discuss the proof of the HyTML algorithm, and its mechanisation in Isabelle. HyTML is equal to SP$\|$FP where SP and FP are the software slow-path and hardware fast-path components, respectively. Recall that we wish to prove $traces(\text{HyTML}) \subseteq traces(\text{TMS2})$. We prove that TMS2$\|$TMS2 weakly simulates HyTML via Theorem 3, and thus $traces(\text{HyTML}) \subseteq traces(\text{TMS2}\|\text{TMS2})$. By Theorem 2, $traces(\text{TMS2}) = traces(\text{TMS2}\|\text{TMS2})$, and hence the result follows by transitivity of $\subseteq$.

We start by defining environment relations for all the automata involved. The relation for the interference SP receives from FP, $I_{\text{FP}}$ is

$$Id \cup \{(g, g') \bullet (odd(glb) \longrightarrow g = g') \wedge ctr' \geq ctr \wedge glb' = glb \\ \wedge (even(glb) \wedge store \neq store' \longrightarrow ctr' > ctr)\}.$$

In words, the fast-path guarantees that: (1) If $glb$ is odd, then it will not affect the global state at all. (2) If $glb$ is even, then any change to the store implies $ctr$

increased. (3) Even if the store remained the same, *ctr* may still increase, and, (4) The fast path never modifies *glb* (it only subscribes to it).

SP makes a much weaker guarantee to the FP; $I_{\mathrm{SP}}$ guarantees that

$$\{(g, g') \bullet \mathit{ctr}' = \mathit{ctr} \wedge \mathit{glb}' \geq \mathit{glb}\}.$$

In words, this means that the software only guarantees that it will not change the *ctr* variable, and that it only ever increments *glb*.

The interference from other TMS2 components on TMS2 is given by $I_{\mathrm{TMS2}}$, which simply allows new stores to be added to the stores sequence (see **(R1)** in Sect. 3).

The proof that TMS2‖TMS2 weakly simulates HyTML is split into several sub-parts: First, we show weak simulation of both $I_{\mathrm{FP}} \rhd \mathrm{SP}$ and $I_{\mathrm{SP}} \rhd \mathrm{FP}$ against $I_{\mathrm{TMS2}} \rhd \mathrm{TMS2}$. The fast-path proof is much simpler than the slow-path, as the hardware transactional memory abstraction performs most of the fine-grained steps of atomically, which greatly simplifies the verification process. Third, we verify the guarantee conditions from Sect. 4. Fourth, we verify the non-interference properties in Sect. 4.

*Mechanisation.* For HyTM implementations we further specialise interference automata to model the components of a hybrid TM implementation. The set of process identifiers become transaction identifiers, and assuming $L$ and $V$ represent the set of all addresses and values, the set of external actions of a transactional automaton $A$ are fixed, and given by:

$$\begin{aligned} \mathit{external}_T = {} & \{\mathbf{Begin_I}, \mathbf{Begin_R}, \mathbf{Commit_I}, \mathbf{Commit_R}, \mathbf{Abort}, \mathbf{Write_R}\} \\ & \cup \{\mathbf{Read_I}(a) \mid a \in L\} \cup \{\mathbf{Read_R}(v) \mid v \in V\} \\ & \cup \{\mathbf{Write_I}(a, v) \mid (a, v) \in L \times V\} \end{aligned}$$

As mentioned in Sect. 2, we base our implementation of the underlying hardware transactional memory on Intel's TSX extensions. Therefore, we implement transitions for the `XBegin`, `XEnd` and `XAbort` actions within the hardware automaton. We assume that each hardware transaction is equipped with read and write sets representing the values held in the local processors cache. A simple validation predicate which checks if the values in the read and write set match those in main memory models the cache line invalidation used in the actual hardware. While this validation is more fine-grained than what the actual hardware can do (as it works on the level of cache lines), because the fast path automaton can abort non-deterministically at anytime, all the possible behaviour of the hardware is captured and shown to be opaque. Overly coarse-grained validation might force us to abort when the hardware could succeed, so we err on the side of caution. This behaviour should be generic enough to capture the behaviour of any reasonable hardware TM implementation, not just Intel's TSX. In particular, we do not assume that non-transactional reads and writes can occur within hardware transactions.

*Proof in Isabelle.* For full-details of our proofs, we refer the interested reader to our Isabelle theories. Here, we briefly comment on the complexity of our mechanisation. In Isabelle, formalising and proving the correctness of the TML slow-path required about 2900 lines, while formalising and proving the correctness of the hardware fast-path required around 600 lines. Proving the non-interference and guarantee conditions required only 450 lines; with the non-interference conditions taking 300 lines and the guarantee conditions requiring only around 70 lines. The formalisation of the transactional automata and requisite theorems took around 2000 lines of Isabelle. Although these are not perfect metrics, they show that the majority of the work was in proving that both HyTM paths satisfy TMS2. Once these individual proofs were completed, bringing the proofs together was fairly comparatively straightforward once the necessary theorems had been set up.

Proving that both HyTM paths are TMS2 is fairly mechanical, and involves detailed line-by-line simulations—showing that every possible step preserves the simulation relation even under interference from every other possible step. Our method enabled adapting our existing work verifying software TML and adapting it to the HyTM case. For simulation proofs of this nature, the number of subgoals grows geometrically with the number of lines in the algorithm, whereas the non-interference conditions only grow linearly in the modular case. However, we believe that both the conceptual benefits of splitting the proof into its logical sub-components, as well as the ability to re-use existing proofs are the main benefits to modularisation.

Our experience with Isabelle for these proofs was very positive. The powerful tools and tactics within Isabelle were very useful for automating many of the cases produced by the simulation rules.

## 8 Conclusion

In this paper we have developed a fully mechanised modular proof method for verifying opacity of HyTM algorithms. Verification of opacity has received considerable interest in recent years (see e.g., [5,13]). We leverage a simulation-based approach against the TMS2 specification [7] as well as the known result that TMS2 is itself opaque [15]. Our method supports adapting existing proofs of opacity (via TMS2) for both the fast- and slow-path into a HyTM system with only minor modifications to such existing proofs.

We develop the novel notion of interference automata, as well as notions of parallel composition and weak simulation for them. These concepts give us a proof method for combining weak simulations on individual interference automata into a single proof of trace refinement for their parallel composition. All of our meta theory has been checked using the Isabelle theorem prover. To show applicability of our methodology in the context of HyTM algorithms, we develop a novel hybrid extension to Dalessandro *et al.*'s TML [4], where we apply a 2-counter subscription mechanism [3]. Our new algorithm allows more concurrency than the original TML as it allows parallel hardware writers.

We conjecture the possibility of further optimisations to the algorithm by removing redundant checks on *glb* and *ctr* in the slow-path read operation if $loc_t$ is odd. It may also be possible to replace the *dccs* operation by first acquiring a local value of *ctr* before acquiring the mutex lock *glb* using a compare and swap and then checking if the local value of *ctr* is still valid. However, we have chosen to present a conceptually simpler algorithm that nevertheless demonstrates our proof method. There are more complex HyTMs [2,3,18], some with more than two types of transactions; we leave verification of these for future work.

**Acknowledgements.** We thank Simon Doherty for his helpful comments on this work. Funding is provided by EPSRC grant EP/N016661/1.

## References

1. Armstrong, A., Dongol, B.: Isabelle files for modularising opacity verification for hybrid transactional memory (2016). https://figshare.com/articles/Isabelle_files_for_verification_of_a_hybrid_transactional_mutex_lock/4868351
2. Calciu, I., Gottschlich, J., Shpeisman, T., Pokam, G., Herlihy, M.: Invyswell: a hybrid transactional memory for Haswell's restricted transactional memory. In: PACT, pp. 187–200. ACM, New York (2014)
3. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M.L., Spear, M.F.: Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. SIGPLAN Not. **46**(3), 39–52 (2011)
4. Dalessandro, L., Dice, D., Scott, M., Shavit, N., Spear, M.: Transactional mutex locks. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 2–13. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15291-7_2
5. Derrick, J., Dongol, B., Schellhorn, G., Travkin, O., Wehrheim, H.: Verifying opacity of a transactional mutex lock. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 161–177. Springer, Cham (2015). doi:10.1007/978-3-319-19249-9_11
6. Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Proving opacity of a pessimistic STM. In: Jiménez, E. (ed.) OPODIS (2016, to appear)
7. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Asp. Comput. **25**(5), 769–799 (2013)
8. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Chatterjee, S., Scott, M.L. (eds.) PPOPP, pp. 175–184. ACM (2008)
9. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, San Rafael (2010)
10. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory. Synthesis Lectures on Computer Architecture, 2nd edn. Morgan & Claypool Publishers, San Rafael (2010)
11. Intel: Intel 64 and IA-32 Architectures Software Developers Manual (2016)
12. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
13. Lesani, M.: On the correctness of transactional memory algorithms. Ph.D. thesis, UCLA (2014)
14. Lesani, M., Luchangco, V., Moir, M.: A framework for formally verifying software transactional memory algorithms. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 516–530. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32940-1_36

15. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place. In: Workshop on the Theory of Transactional Memory (2012)
16. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, Burlington (1996)
17. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC, pp. 137–151. ACM (1987)
18. Matveev, A., Shavit, N.: Reduced hardware NOrec: a safe and scalable hybrid transactional memory. SIGPLAN Not. **50**(4), 59–71 (2015)
19. Paulson, L.C.: Isabelle - A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994). (with a contribution by Nipkow, T.)
20. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, New York (2001)