Specifying Transactional Memories with Nontransactional Operations

Mohsen Lesani
UCLA
mohsen.lesani@gmail.com

Victor Luchangco   Mark Moir
Oracle Labs
{victor.luchangco, mark.moir}@oracle.com

1. Introduction

Although transactional memory (TM) is a promising approach for synchronizing shared-memory concurrent programs, it will not exist alone: real systems will provide a variety of synchronization mechanisms, and TM must interact properly with them. Therefore, a full specification for TM must specify how it interacts with nontransactional operations.

One possibility is that there is no interaction: memory managed by the TM is statically determined and accessed only through the TM interface. However, TM-mediated access is more expensive than unmediated access, particularly with software implementations, so programmers may want unmediated access to memory when they know it will not be accessed concurrently by the TM. Also, programmers may need to use existing libraries, or integrate their TM-based programs into existing applications. In many cases, it would be onerous, even infeasible, to rewrite the existing code. Thus, we want a TM specification for implementations that allow *uninstrumented* nontransactional access.

In this paper, we extend TMS1 [3] to allow nontransactional operations. We call our extension NTMS1. TMS1 is designed to specify requirements for a TM runtime library in unmanaged languages such as C or C++, running on a system that may support other mechanisms for coordinating threads. Like most work on specifying TM semantics (e.g., [4, 5, 8]), TMS1 specifies semantics only for transactional operations, implicitly assuming that memory accessed with transactions is not accessed outside transactions. Because threads executing transactions might coordinate using other mechanisms between transactions, TMS1 requires transactions to respect the "external order" of transactions. NTMS1 follows TMS1 in all these respects.

Prior work on verifying TM with nontransactional operations [2] required *strong atomicity*: nontransactional operations are equivalent to "mini-transactions" that cannot abort. Because such an implementation cannot be achieved in general without instrumenting nontransactional operations, we do not follow this. Rather, following Adve [1], NTMS1 defines transactional semantics for programs without *data races*, and allows arbitrary behavior for programs with such races.

As with TMS1, we want NTMS1 to afford TM implementors as much flexibility as possible while still guaranteeing transactional semantics, which we informally define as each transaction appearing to execute one at a time without any interleaved operations by other threads. Handling active and aborted transactions properly involves subtleties [3, 4, 5], even in the absence of nontransactional operations. We discuss these subtleties, and further ones introduced by nontransactional operations, in later sections. We have proved that a data-race-free program (as defined later) cannot distinguish a transactional memory implementation satisfying NTMS1 from one that provides strong atomicity.

2. Preliminaries

I/O automata We formalize our specifications as *I/O automata* (IOAs) [7]. An IOA A specifies a set *states*(A) of states, a nonempty subset *start*$(A) \subseteq$ *states*(A) of start states, a set *acts*(A) of actions, a signature *sig*$(A) = ($*in*$(A),$ *out*$(A),$ *internal*$(A))$ that partitions *acts*(A) into input, output and internal actions, and a transition relation *trans*$(A) \subseteq$ *states*$(A) \times$ *acts*$(A) \times$ *states*(A). The *external interface* of A is defined by its external actions $($*in*$(A),$ *out*$(A))$. An action a is *enabled* in a state s if $(s, a, s') \in$ *trans*(A) for some $s' \in$ *states*(A). An IOA must be *input-enabled*: every input action must be enabled in every state.

An *execution fragment* of A is a sequence $s_0 a_1 s_1 \ldots$ of alternating states and actions such that $(s_{i-1}, a_i, s_i) \in$ *trans*(A) for all i. An *execution* is an execution fragment with $s_0 \in$ *start*(A). The *trace* of an execution fragment is the subsequence of its external actions. The traces of A are the traces of its executions; we denote the set of such traces by *traces*(A). An IOA A *implements* another IOA B if *traces*$(A) \subseteq$ *traces*(B).

Sequential semantics of objects NTMS1 is defined for an arbitrary object type, which specifies the *interface* and *sequential semantics* of an object. The interface consists of a set $\mathcal{I}$ of *operation invocations* and a set $\mathcal{R}$ of *operation responses*. An *operation* is an invocation-response pair, and a *sequential history* is a sequence of operations. The sequential semantics is a set of *legal sequential histories*, specified by a predicate *legal* on sequential histories.

To specify data races, an object type also specifies a *conflict relation*, a symmetric binary relation on operation invocations, represented by a predicate *conflict*.

TM interface Given a set $\mathcal{T}$ of transaction identifiers and a set $\mathcal{N}$ for nontransactional operation identifiers, the external interface of a transactional memory system supporting an object type with operation invocations $\mathcal{I}$ and operation responses $\mathcal{R}$ is:

Input actions	Output actions
begin_t for $t \in \mathcal{T}$	$\mathsf{beginOk}_t$ for $t \in \mathcal{T}$
$\mathsf{tInv}_t(i)$ for $t \in \mathcal{T}, i \in \mathcal{I}$	$\mathsf{tResp}_t(r)$ for $t \in \mathcal{T}, r \in \mathcal{R}$
commit_t for $t \in \mathcal{T}$	$\mathsf{commitOk}_t$ for $t \in \mathcal{T}$
cancel_t for $t \in \mathcal{T}$	abort_t for $t \in \mathcal{T}$
$\mathsf{nInv}_n(i)$ for $n \in \mathcal{N}, i \in \mathcal{I}$	$\mathsf{nResp}_n(r)$ for $n \in \mathcal{N}, r \in \mathcal{R}$

We call the input actions *invocations* and the output actions *responses*.

Clients using the TM have the opposite interface (i.e., with invocations as output actions and responses as input actions). Although a TM implementation that supports uninstrumented transactional operations would not actually see the invocations and responses of such operations, we include them in the TM interface to specify the interaction between transactions and nontransactional operations.

Well-formedness A TM implementation and its clients usually obey some simple well-formedness conditions, such as that a response other than abort occurs only in response to a corresponding invocation (e.g., $\mathsf{beginOk}_t$ in response to begin_t)—abort may be the response to any invocation—and that a client thread does not issue an invocation while it has one pending, it does not issue multiple nontransactional operation invocations with the same identifier, and it does not issue any invocation for transaction *t* before it issues begin_t or after it receives $\mathsf{commitOk}_t$ or abort_t. (As discussed in Section 4, well-formedness may be violated by racy programs because we allow arbitary behavior in the presence of races.)

For a well-formed execution and a transaction *t*, we say that begin_t is the *beginning invocation*, and $\mathsf{commitOk}_t$ or abort_t is the *final response*, if it exists in the execution, and the *operation sequence* of *t* is the sequence of operations (i.e., operation invocation-response pairs) it has invoked and received a (non-abort) response to. For convenience, the invocation and response of a nontransactional operation *n* are its *beginning invocation* and *final response* respectively, and its *operation sequence* is the empty sequence unless *n* has completed, in which case it has a single operation with *n*'s operation invocation and response. We also define the *external order* to be a partial order on $\mathcal{X} = \mathcal{T} \cup \mathcal{N}$ that orders *x* before x' in the external order if the final response of *x* precedes the beginning invocation of x' in the execution. We say that a sequence $\sigma \in \mathcal{X}^*$ with no duplicate elements is a *legal serialization* if σ respects the external order and the concatenation of the operation sequences of the elements of σ (in the order they appear in σ) is a legal sequential history of the TM's object type.

Serial executions Informally, we want a programmer using transactions to be able to reason about the correctness of a program as though each transaction executes atomically. We make this precise by defining a *serial execution* as a well-formed execution in which all events associated with each transaction occur consecutively, and every transaction and nontransactional operation, except possibly the last one, has completed (i.e., committed or aborted for a transaction, responded for a nontransactional operation). Thus, the external order of a serial execution totally orders the transactions and nontransactional operations that occur in the execution.

In a serial execution, we say that a transaction or nontransactional operation *sees* itself and every committed transaction and nontransactional operation that occurs before itself. A transaction or nontransactional operation is *legal* in a serial execution if the sequence of transactions and nontransactional operations it sees (in the order that they appear in the serial execution) is a legal serialization. A serial execution is *legal* if every transaction and nontransactional operation is legal in that execution.

3. TMS1

The TMS1 correctness condition [3] is designed to maximize implementation flexibility subject to the following constraints:

- no transaction observes partial effects of any other transaction;
- there must always be a legal serialization that includes all committed transactions and no aborted transactions; and
- the behavior observed by each transaction, even one that subsequently aborts, is consistent with being part of some serial execution.

To specify the first condition, we say that a transaction is *visible* if it has invoked commit. Before it is visible, the transaction is still active, and may have further effects, so effects it has already had should not be seen by any other transaction.

The second condition corresponds to the usual serializability requirement for transactional systems.

The last condition allows a programmer to reason about the correctness of a program by considering only serial executions. Without it, a transaction might observe a pattern of responses that violates some property assumed by the programmer because it is true in all serial executions, and thus invoke an operation that causes an unrecoverable error (e.g., dividing by zero or accessing memory not allocated to the application). In an unmanaged language, this can result, for example, in program termination before the transactional memory implementation can abort the offending transaction. Because the only purpose of this condition is to avoid unrecoverable errors, different transactions need not be "justified" by the same serial execution. Only transactions that commit need to be consistent with each other: while a transaction is active, no other transaction can infer what operations it has invoked; if it aborts, no other transaction can ever infer them.

TMS1 is defined as an IOA that tracks, for each transaction, the sequence of operations it invoked, and responses generated, and whether the transaction committed or aborted. It also maintains the external order on transactions. The interesting aspects of this automaton are captured by *validCommit*, *validFail* and *validResp*, three "validation" conditions on responses (no validation condition is needed for $\mathsf{beginOk}_t$) that ensure the TM satisfies the constraints above.

The validation conditions for committing and aborting are straightforward (though it may be surprising that there is such a condition for aborting; see [3] for a discussion of this point), but *validResp* is tricky because the response to an operation of an active transaction may reflect the effects of an aborted transaction, provided that the two transactions were concurrent and the aborted transaction invoked commit before it aborted. (To conform to the first constraint above, a transaction must not observe the effects of any transaction that has not invoked commit.) However, if it sees the effect of any transaction *t*, then it must also see the effect of any committed transaction that precedes *t* in the external order and it must *not* see the effect of any aborted transaction that precedes *t* in the external order, because *t* must see the effects of the former and not the latter. We capture this requirement with the notion of an *externally consistent prefix*. That is, a set S of transaction identifiers in a state of TMS1 is an *externally consistent prefix* if for all $t \in \mathcal{T}$ and $t' \in S$, if *t* precedes t' in the external order, then $t \in S$ if and only if *t* is committed.

With this notion, the validation condition for $\mathsf{tResp}_t(r)$ in response to an invocation *i* is that there must be a subset of visible transactions that, together with *t*, form an externally consistent prefix with a legal serialization after the new operation has been added to the operation sequence of *t*.

4. NTMS1

To extend TMS1 to support nontransactional operations, we adapt the various definitions and conditions used by TMS1 to apply to $\mathcal{X} = \mathcal{T} \cup \mathcal{N}$ rather than just $\mathcal{T}$. We also define *data races*, and allow NTMS1 to exhibit arbitrary behavior when they occur. The resulting IOA appears in Figure 1. (Due to space constraints, this IOA is really the composition of NTMS1 with an IOA modeling data-race-free clients, which we describe after NTMS1.)

The adaptation to use $\mathcal{X}$ is straightforward, and much of it can already be seen in Section 2. As in TMS1, we maintain the "status" of each transaction, indicating whether it has not yet started, is "ready" to invoke an operation, has a pending invocation (and what kind), or has committed or aborted. It also maintains the operation invocation if it has an operation pending, the operation sequence (i.e., the operations for which it already has a response), and a flag indicating whether it invoked commit (to determine

State variables

$\mathit{status}[\mathcal{X}]$: $\{\mathsf{notStarted}, \mathsf{beginning}, \mathsf{ready}, \mathsf{opPending}, \mathsf{committing}, \mathsf{canceling}, \mathsf{committed}, \mathsf{aborted}\}$; initially all $\mathsf{notStarted}$
$\mathit{ops}[\mathcal{X}]$: $(\mathcal{I} \times \mathcal{R})^*$ (i.e., sequence of operations); initially all empty
$\mathit{opInv}[\mathcal{X}]$: $\mathcal{I}$; initially arbitrary
$\mathit{invokedCommit}[\mathcal{T}]$: Boolean; initially $\mathbf{false}$

$\mathit{extOrder}$: binary relation on $\mathcal{X}$; initially empty
tHavoc: Boolean; initially $\mathbf{false}$
cHavoc: Boolean; initially $\mathbf{false}$

Actions

begin_t, $t \in \mathcal{T}$
Pre: $\mathit{cHavoc} \lor \mathit{status}[t] = \mathsf{notStarted}$
Eff: $\mathit{status}[t] \leftarrow \mathsf{beginning}$
$\quad\mathit{extOrder} \leftarrow \mathit{extOrder} \cup (\mathit{DX} \times \{t\})$

$\mathsf{beginOk}_t$, $t \in \mathcal{T}$
Pre: $\mathit{tHavoc} \lor \mathit{status}[t] = \mathsf{beginning}$
Eff: $\mathit{status}[t] \leftarrow \mathsf{ready}$

$\mathsf{tInv}_t(i)$, $t \in \mathcal{T}, i \in \mathcal{I}$
Pre: $\mathit{cHavoc} \lor (\mathit{status}[t] = \mathsf{ready} \land \neg \mathit{tRace}(t, i))$
Eff: $\mathit{status}[t] \leftarrow \mathsf{opPending}$
$\quad\mathit{opInv}[t] \leftarrow i$

$\mathsf{tResp}_t(r)$, $t \in \mathcal{T}, r \in \mathcal{R}$
Pre: $\mathit{tHavoc} \lor (\mathit{status}[t] = \mathsf{opPending} \land \mathit{tValidResp}(t, \mathit{opInv}[t], r))$
Eff: $\mathit{status}[t] \leftarrow \mathsf{ready}$
$\quad\mathit{ops}[t] \leftarrow \mathit{ops}[t] \circ (\mathit{opInv}[t], r)$

commit_t, $t \in \mathcal{T}$
Pre: $\mathit{cHavoc} \lor \mathit{status}[t] = \mathsf{ready}$
Eff: $\mathit{status}[t] \leftarrow \mathsf{committing}$
$\quad\mathit{invokedCommit}[t] \leftarrow \mathbf{true}$

$\mathsf{commitOk}_t$, $t \in \mathcal{T}$
Pre: $\mathit{tHavoc} \lor (\mathit{status}[t] = \mathsf{committing} \land \mathit{validCommit}(t))$
Eff: $\mathit{status}[t] \leftarrow \mathsf{committed}$

cancel_t, $t \in \mathcal{T}$
Pre: $\mathit{cHavoc} \lor \mathit{status}[t] = \mathsf{ready}$
Eff: $\mathit{status}[t] \leftarrow \mathsf{canceling}$

abort_t, $t \in \mathcal{T}$
Pre: $\mathit{tHavoc} \lor (\mathit{status}[t] \in \{\mathsf{beginning}, \mathsf{opPending}, \mathsf{committing}, \mathsf{canceling}\} \land \mathit{validFail}(t))$
Eff: $\mathit{status}[t] \leftarrow \mathsf{aborted}$

$\mathsf{nInv}_n(i)$, $n \in \mathcal{N}, i \in \mathcal{I}$
Pre: $\mathit{cHavoc} \lor (\mathit{status}[n] = \mathsf{notStarted} \land \neg \mathit{nRace}(i))$
Eff: $\mathit{status}[n] \leftarrow \mathsf{opPending}$
$\quad\mathit{opInv}[n] \leftarrow i$

$\mathsf{nResp}_n(r)$, $n \in \mathcal{N}, r \in \mathcal{R}$
Pre: $\mathit{tHavoc} \lor (\mathit{status}[n] = \mathsf{opPending} \land \mathit{nValidResp}(n, \mathit{opInv}[t], r))$
Eff: $\mathit{status}[n] \leftarrow \mathsf{committed}$
$\quad\mathit{ops}[t] \leftarrow (\mathit{opInv}[t], r)$

$\mathsf{observeIncorrectTM}$
Pre: $\neg \mathit{correctTM}$
Eff: $\mathit{cHavoc} \leftarrow \mathbf{true}$

$\mathsf{observeRace}$
Pre: $\exists t \in \mathcal{T}, n \in \mathcal{N},\ \mathit{tnRace}(t, n) \lor \exists n, n' \in \mathcal{N},\ \mathit{nnRace}(n, n')$
Eff: $\mathit{tHavoc} \leftarrow \mathbf{true}$

Derived state variables and predicates

$$
\begin{aligned}
\mathit{DX} &\triangleq \{x \mid \mathit{status}[x] \in \{\mathsf{committed}, \mathsf{aborted}\}\} \\
\mathit{CX} &\triangleq \{x \mid \mathit{status}[x] = \mathsf{committed}\} \\
\mathit{CPX} &\triangleq \{x \mid \mathit{status}[x] = \mathsf{committing}\} \\
\mathit{VX} &\triangleq \{t \mid \mathit{invokedCommit}[t]\} \cup \{n \mid \mathit{status}[n] = \mathsf{committed}\} \\
\mathit{extConsPrefix}(S) &\triangleq \forall x, x' \in \mathcal{X},\ x' \in S \land (x, x') \in \mathit{extOrder} \implies (x \in S \iff \mathit{status}[x] = \mathsf{committed}) \\
\mathit{validCommit}(t) &\triangleq \exists S \subseteq \mathit{CPX}, \exists \sigma \in \mathit{ser}(\mathit{CX} \cup S, \mathit{extOrder}),\ t \in S \land \mathit{legal}(\circ_k(\mathit{ops}[\sigma_k])) \\
\mathit{validFail}(t) &\triangleq \exists S \subseteq \mathit{CPX}, \exists \sigma \in \mathit{ser}(\mathit{CX} \cup S, \mathit{extOrder}),\ t \notin S \land \mathit{legal}(\circ_k(\mathit{ops}[\sigma_k])) \\
\mathit{tValidResp}(t, i, r) &\triangleq \exists S \subseteq \mathit{VX}, \exists \sigma \in \mathit{ser}(S, \mathit{extOrder}),\ \mathit{extConsPrefix}(S \cup \{t\}) \land \mathit{legal}(\circ_k(\mathit{ops}[\sigma_k]) \circ \mathit{ops}[t]) \circ (i, r)) \\
\mathit{nValidResp}(n, i, r) &\triangleq \exists S \subseteq \mathit{CPX}, \exists \sigma \in \mathit{ser}(\mathit{CX} \cup S \cup \{n\}, \mathit{extOrder}),\ \mathit{legal}(\circ_k(\mathit{ops}'[\sigma_k])) \text{ where } \mathit{ops}'[n] = (i, r) \text{ and } \mathit{ops}'[x] = \mathit{ops}[x] \text{ for } x \neq n \\
x \parallel x' &\triangleq x \neq x' \land \mathit{status}[x] \neq \mathsf{notStarted} \land \mathit{status}[x'] \neq \mathsf{notStarted} \land (x, x') \notin \mathit{extOrder} \land (x', x) \notin \mathit{extOrder} \\
\mathit{tnRace}(t, n) &\triangleq t \parallel n \land (\exists (i, r) \in \mathit{ops}[t],\ \mathit{conflict}(i, \mathit{opInv}[n]) \lor \mathit{status}[t] = \mathsf{opPending} \land \mathit{conflict}(\mathit{opInv}[t], \mathit{opInv}[n])) \\
\mathit{nnRace}(n, n') &\triangleq n \parallel n' \land \mathit{conflict}(\mathit{opInv}[n], \mathit{opInv}[n']) \\
\mathit{tRace}(t, i) &\triangleq \exists n \in \mathcal{N},\ \mathit{status}[n] \neq \mathsf{notStarted} \land (n, t) \notin \mathit{extOrder} \land \mathit{conflict}(\mathit{opInv}[n], i) \\
\mathit{nRace}(i) &\triangleq (\exists x \in \mathcal{X},\ \mathit{status}[x] = \mathsf{opPending} \land \mathit{conflict}(\mathit{opInv}[x], i)) \\
&\qquad \lor (\exists t \in \mathcal{T},\ (\mathit{status}[t] \notin \{\mathsf{notStarted}, \mathsf{committed}, \mathsf{aborted}\} \land \exists (i', r) \in \mathit{ops}[t],\ \mathit{conflict}(i', i)) \\
\mathit{correctTM} &\triangleq (\exists S \subseteq \mathit{CPX}, \exists \sigma \in \mathit{ser}(\mathit{CX} \cup S, \mathit{extOrder}),\ \mathit{legal}(\circ_k(\mathit{ops}[\sigma_k]))) \\
&\qquad \land (\forall t \in \mathcal{T},\ \exists S \subseteq \mathit{VX},\ \exists \sigma \in \mathit{ser}(S, \mathit{extOrder}),\ \mathit{extConsPrefix}(S \cup \{t\}) \land \mathit{legal}(\circ_k(\mathit{ops}[\sigma_k]) \circ \mathit{ops}[t]))
\end{aligned}
$$

Figure 1. $\mathsf{NTMS1} \times \mathsf{RFC}$ with transaction identifiers $\mathcal{T}$ and nontransactional operation identifiers $\mathcal{N}$ (and $\mathcal{X} = \mathcal{T} \cup \mathcal{N}$), for object type with operation invocations $\mathcal{I}$, operation responses $\mathcal{R}$, sequential semantics defined by *legal*, and conflict relation *conflict*. Some notation requires explanation: $\mathit{ser}(S, \mathit{extOrder})$ is the set of all serializations of S that respect *extOrder*; that is, $\sigma \in \mathit{ser}(S, \mathit{extOrder})$ if and only if σ contains every element of S exactly once and for all $x, x' \in S$, x precedes x' in σ whenever $(x, x') \in \mathit{extOrder}$. The $\circ$ operator is for concatenation, and $\circ_k(\mathit{ops}[\sigma_k]) = \mathit{ops}[\sigma_1] \circ \cdots \circ \mathit{ops}[\sigma_m]$, where $m = |\sigma|$. Thus, $\sigma \in \mathit{ser}(S, \mathit{extOrder}) \land \mathit{legal}(\circ_k(\mathit{ops}[\sigma_k]))$ means that σ is a legal serialization of S, as defined in Section 2.

whether the transaction is visible). We maintain similar information, except for the flag, for each nontransactional operation, but its status can only be that it has not yet started, has a pending operation, or is done. (For convenience, we use committed as the status of a completed nontransactional operation.) We also maintain the external order. These variables are updated in the obvious way in NTMS1 (as they are in TMS1).

The validation conditions for transactional invocations in NTMS1 are essentially the same as in TMS1, except that a nontransactional operation is visible when it is done though it never invokes commit. We rename the validation condition for a transactional operation response *tValidResp*, to distinguish it from *nValidResp*, a new validation condition for the response to a nontransactional operation. This new validation condition is more similar to *validCommit* than to *tValidResp* because the response to a nontransactional operation invocation is immediately visible, and so is akin to committing a transaction: it must be consistent with committed transactions and other completed nontransactional operations.

We define data races using the conflict relation of the object type, which must be a symmetric relation. Specifically, two nontransactional operations *race* if they overlap in time and their operation invocations conflict according the conflict relation; this is modeled by the predicate *nnRace*. Transactions never race with each other (the TM implementation coordinates their accesses), but a transaction *races* with a nontransactional operation if they overlap in time and any operation invoked by the transaction conflicts with the nontransactional operation invocation; this is modeled by the predicate *tnRace*.

To model the freedom from constraints in the presence of data races, we augment NTMS1 with a boolean flag *tHavoc* and an internal action observeRace that sets *tHavoc* to `true` if there is a data race, and we allow any response once *tHavoc* is set (i.e., we add *tHavoc* as a disjunct to the precondition of every response). Thus, once a data race is detected, NTMS1 may exhibit arbitrary behavior, possibly violating even well-formedness.

To use NTMS1 effectively, a program must not produce any data races. We call such a program *data-race-free*. Although it is easy in principle to determine whether a given execution has a data race, determining whether a *program* is data-race-free is tricky, because what operations a program invokes may depend in part on the responses it receives. Thus, a faulty TM implementation may cause a correct (i.e., data-race-free) program to be racy.

We address this issue by defining another automaton RFC ("race-free clients"), which specifies precisely what it means for the clients to be data-race-free. Because RFC models the clients, invocations are its output actions and responses are its input actions. In addition to the basic well-formedness conditions, RFC adds a conjunct to the preconditions of $\mathsf{tInv}_t(i)$ and $\mathsf{nInv}_n(i)$ to ensure that the new operation invocation does not introduce a data race (using predicates *tRace* and *nRace*). RFC maintains exactly the same state as NTMS1 except that instead of *tHavoc*, it has its own *cHavoc* flag, which it sets when it detects that the TM has given an incorrect response. This is modeled by the observeIncorrectTM internal action. (Note that we use "detect" here in an abstract sense: once a TM has given an incorrect response, RFC allows arbitrary behavior. The program might not actually know that the TM was incorrect.)

As mentioned above, the automaton presented in Figure 1 is really the composition of NTMS1 and RFC. It is easy to extract the component automata, because all state variables other than *tHavoc* (which is only in NTMS1) and *cHavoc* (which is only in RFC) are updated in exactly the same way by both NTMS1 and RFC, and these flags are updated only by observeRace (for *tHavoc*) and observeIncorrectTM (for *cHavoc*), which are internal actions of NTMS1 and RFC respectively. All the other actions are external actions, and each is an input action of one of NTMS1 or RFC and an output action of the other, so the precondition goes with the automaton for which it is an output action (recall that IOA must be input-enabled, so their input actions do not have preconditions).

Using a framework we have developed in PVS for verifying TM [6], we have formalized these automata, along with an automaton that specifies strong atomicity, and we have proven that the composite IOA NTMS1 $\times$ RFC implements the automaton for strong atomicity. This implies that data-race-free clients cannot distinguish a TM implementation satisfying NTMS1 from one that guarantees strong atomicity.

One possibly surprising aspect of our definitions is that the conflict relation of an object type can be any symmetric binary predicate, and imposes no restriction on the sequential semantics (i.e., the set of legal sequential histories). One might think that two operations that do not conflict ought to commute, for some definition of "commute". We expect that this will typically be so. However, it is *not* necessary for our specification: the choice of conflict relation simply shifts the burden between TM implementors and clients. For example, if the conflict relation is empty, there are no data races, so the TM implementation must effectively guarantee strong atomicity. At the other extreme, if every operation conflicts with every operation (including itself), then any nontransactional operation that overlaps either a (nontrivial) transaction or another nontransactional operation forms a data race, so programmers must synchronize all their accesses accordingly.

5. Discussion

Although we intended NTMS1 to restrict TM implementors as little as possible, on further consideration, it could be relaxed by making nontransactional operations visible before they complete. For example, we could use an internal action that effectively does the operation, after which its effects can be seen by transactions and other nontransactional operations. We believe it would be straightforward to make this change.

It may be possible to go further still, and allow a nontransactional operation to be visible immediately upon invocation. This change is trickier because the operation response is not yet fixed, and we must ensure that we won't get "stuck" in a state in which there is no valid response for a nontransactional operation.

Note that NTMS1 does *not* guarantee "privatization-safety": For example, one transaction may write a location x and then invoke commit, and do everything necessary to successfully commit except send the response. If another transaction then privatizes x and accesses it nontransactionally before the first transaction returns, this forms a data race according to our definition. So NTMS1 allows arbitrary behavior from that point on. This is a concern because a motivation for this work is to specify TM for unmanaged languages like C++. However, the transactional constructs being considered for C++ do guarantee privatization-safety. We are thus motivated to consider alternatives that do guarantee privatization-safety.

Unfortunately, we don't know of any precise characterization of privatization-safety. Rather, what exists is an intuition and some examples of programs (or patterns of access) that we want to consider correct (i.e., data-race-free) and so should be guaranteed "transactional semantics". It is difficult to see how a property like privatization-safety could even be stated in the generic context of NTMS1, which treats the entire memory as a single object. Even restricting to TM specifications for read-write memory, the notion of "privatization" suggests some kind of unique ownership or control over some locations in memory, which would require us to model threads or some similar entities to be the owners or controllers of privatized locations, which NTMS1 does not do.

References

[1] Sarita Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993.

[2] Ariel Cohen, Amir Pnueli, and Lenore Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 121–134, 2008.

[3] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.

[4] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 175–184, 2008.

[5] Damien Imbs, Jose de Mendivil, and Michel Raynal. Virtual world consistency: A new condition for STM systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 280–281, 2009.

[6] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR)*, pages 516–530, 2012.

[7] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987.

[8] Michael Scott. Sequential specification of transactional memory semantics. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.