Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 259 (2009) 245–261
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Towards Formally Specifying and Verifying Transactional Memory [1,2]

Simon Doherty and Lindsay Groves[3]

School of Engineering and Computer Science
Victoria University of Wellington
New Zealand

Victor Luchangco and Mark Moir[4]

Sun Microsystems Laboratories
Burlington, MA
USA

Abstract

We describe ongoing work in which we aim to formally specify a correctness condition for transactional memory (TM) called Weakest Reasonable Condition (*WRC*), and to facilitate fully formal and machine-checked proofs that TM implementations satisfy the condition. To precisely define the *WRC* condition, we express it using an I/O automaton. We similarly present another condition, called *PRAG*, which is more restrictive, but more closely reflects intuition about common TM implementation techniques. We sketch a simulation proof that *PRAG* implements *WRC*, allowing ourselves and others to focus more pragmatically on proofs of such implementations. We are working on modeling these conditions in the PVS language so that we can construct and check such proofs precisely and mechanically. We are also working towards proving that some popular TM implementations satisfy the *PRAG* condition, starting with simple coarse-grained versions and refining them to model realistic implementations.

Keywords: Transactional memory, simulation, correctness condition, opacity, virtual worlds consistency.

1 Introduction

Transactional memory (TM) [9] aims to make it significantly easier to develop and maintain concurrent programs that are scalable, efficient, and correct by allowing programmers to specify that a sequence of operations on shared objects should be executed as a *transaction*. A transaction makes the sequence of operations

[1] This work is supported by a donation from Sun Microsystems Laboratories.

[3] Email:{simon.doherty,lindsay}@ecs.vuw.ac.nz
[4] Email:{victor.luchangco,mark.moir}@sun.com

doi:10.1016/j.entcs.2010.01.001

appear to be applied *atomically* (i.e., without interference from concurrent threads, and without concurrent threads observing partial results of the sequence) without specifying the synchronisation mechanism used to achieve such atomicity.

Because TM implementations aim to hide some of the complexity of concurrent programming in system software, it is important that they be correct. We are therefore pursuing a long-term goal of developing formal and machine-checked correctness proofs for TM implementations. We hope to follow an approach that we have already used to verify a range of concurrent algorithms [2,3,5], specifying both the permitted behaviour and the behaviour of the algorithm using I/O automata (IOAs) [11], and using simulation proof techniques [12] to show that an algorithm implements a given specification. We construct these proofs using the PVS verification system [13], so our proofs can be entirely machine-checked.

In this paper, we describe foundational work we are doing to apply this approach to formally verifying TM implementations. In particular, such verification requires a formal description of the legal behaviour of a TM implementation. In our previous work, the algorithms implemented data structures with well-defined, universally accepted semantics, so specifying the permitted behaviours was straightforward. In contrast, transactional memory, does not have universally accepted semantics, but instead many variants. For now, we consider only the simple case in which variables are not shared between transactional and nontransactional code.

Even in this case, there are subtleties: The semantics of committed transactions are straightforward, but what guarantees are provided to transactions that abort? It is important that they observe a consistent view of the memory up until the point at which they abort [4]: a transaction that sees inconsistent state may take steps that cause segmentation violations, etc. For example, a program may maintain an invariant that two variables x and y are never equal, and then divide by $x - y$ within a transaction. If that transaction sees inconsistent state in which x and y are equal, it will cause a divide-by-zero error. Although in some cases, it is possible to "sandbox" transactions so that such errors can be hidden, this is not always possible, especially in unmanaged languages such as C or C++.

In any case, our initial focus is on conditions that require *all* reads performed by any transaction to be consistent. We provide formal descriptions of such conditions, using IOAs to facilitate our approach to achieving formal, machine-checked correctness proofs. Other researchers have specified TM correctness conditions that make guarantees about the consistency of values read by aborted transactions [8,10], with the same motivation. Our work differs in the precise meaning of "consistent", and in the methods we use to express these conditions and prove that TM implementations satisfy them. For example, some previous work on showing that implementations satisfy such properties uses a combination of abstraction and model checking [6,7]. This approach is limited to implementations that share certain structural properties, and it is challenging to prove that an implementation has these properties. Furthermore, the limitations of model checking necessitate concessions both in the models and in the correctness conditions. Other related work [1] is closer in spirit and approach to ours, but uses a correctness condition that is too strong to be used for many popular TM implementations.

In defining a correctness condition for TM, we have several goals beyond providing an unambiguous and precise definition that supports formal machine-checkable proofs: The condition must make sufficient semantic guarantees to be useful to programmers using TM. It should be easy to understand and reason about, so that researchers and implementors can prove that TM implementations satisfy the condition. And it should be as permissive as possible to avoid arbitrarily excluding implementation techniques, including ones not yet invented.

There is tension between these goals: the generality of a highly permissive correctness condition makes the condition harder to understand, while admitting additional behaviours that are exhibited by few, if any, real implementations. We deal with this tension by specifying multiple correctness conditions using IOAs, which support hierarchical reasoning via simulation proofs [12], so that an automaton specifying a TM correctness condition can be used both as a specification for a specific TM implementation and as an implementation of a more permissive condition.

In this paper, we present two conditions: *WRC*—which stands for Weakest Reasonable Condition—is very general and permissive. This name reflects our motivation and should not be overinterpreted, given that what is reasonable depends to some extent on context. *PRAG* is more pragmatic: it is less general, less permissive, and closer to the intuition of how most existing TM runtimes work. We sketch a proof that *PRAG* implements *WRC*. Thus researchers can prove that their implementation implements *PRAG*, and conclude that it implements *WRC*.

In Section 2 we describe I/O automata, proof techniques, TM interfaces, and notation used in the rest of the paper. Sections 3 and 4 introduce the *WRC* correctness condition and relate it to two previous conditions. We present the *PRAG* condition in Section 5, and then sketch a simulation proof that it implies the *WRC* condition in Section 6. We briefly discuss our ongoing and future work in Section 7.

2 Preliminaries

This section provides background on how we express TM correctness conditions, how we model TM implementations, and how we prove relationships between them.

2.1 I/O automata

We use *input/output automata* (IOAs) [11] to express TM correctness conditions and to model TM implementations. An IOA A is a labelled transition system that consists of: a set $states(A)$ of states; a nonempty set $start(A) \subseteq states(A)$ of start states; a set $acts(A)$ of actions; a signature $sig(A) = (external(A), internal(A))$, which partitions $acts(A)$; and a transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$. (We do not partition $external(A)$ into *input* and *output* actions for this paper because we do not need to compose automata.) We describe a transition relation using a *precondition* (a predicate on states) and an *effect* (a set of assignments to variables) for each action.

An *execution fragment* of A is a sequence $s_0, a_1, s_1, \ldots$ of alternating states and actions of A, such that $(s_{k-1}, a_k, s_k) \in \mathit{trans}(A)$ for all k; a finite sequence must end with a state. An *execution* is an execution fragment with $s_0 \in \mathit{start}(A)$. The subsequence of external actions in an execution fragment is called its *trace*, and represents its externally visible *behaviour*. The traces of an automaton A are the traces of its executions; we denote the set of such traces by $\mathit{traces}(A)$. For an "abstract" automaton A, modelling a specification, and a "concrete" automaton C, modelling an implementation, C *implements* A iff $\mathit{traces}(C) \subseteq \mathit{traces}(A)$: every behaviour of the implementation is allowed by the specification.

2.2 Simulation proofs

One way to prove that C implements A is via a *forward simulation* [12], which is a relation between $\mathit{states}(C)$ and $\mathit{states}(A)$ such that every start state of C is related to some start state of A, and for every step $(s, a, s') \in \mathit{trans}(C)$ and every $u \in \mathit{states}(A)$ that is related to s by the forward simulation, there is an execution fragment of A starting from u and ending in a state u' such that (i) u' and s' are related by the forward simulation, and (ii) the execution fragment has the same trace as the step of C. That is, if a is an internal action, then the execution fragment has only internal actions, and if a is an external action, then the execution fragment contains that action and no other external actions. With a forward simulation from C to A, we can prove inductively that any trace of C is also a trace of A: given an execution of C, we can construct an execution of A with the same trace by choosing a start state of A related to the start state of C's execution by the forward simulation, and then for every step of C in turn, extending the execution of A with the execution fragment required by the forward simulation. (*Backward simulations* are also important for our work (see Section 7), but are not used in this paper.)

2.3 Objects and their sequential semantics

To state and prove properties of TM implementations that support general objects, we need to formally define their interface and sequential semantics.

The interface for an object $\mathcal{O}$ consists of a set $\mathcal{I}_\mathcal{O}$ of possible *invocations* and a set $\mathcal{R}_\mathcal{O}$ of possible *responses*. An invocation-response pair is an *operation* of the object. The *sequential semantics* of an object $\mathcal{O}$ specifies which sequences of operations (i.e., elements of $(\mathcal{I}_\mathcal{O} \times \mathcal{R}_\mathcal{O})^*$) are *legal sequential histories*.

Most TM implementations assume a specific type of object called a *read-write memory*. For this reason, our *PRAG* automaton is specialised for a read-write memory. A read-write memory maps a set L of *locations* to a set V of *values*. When used by a set $\mathcal{T}$ of transactions, its interface is:

$$\mathcal{I}_{\mathrm{RW}} = \{\mathsf{inv}_t(\mathit{read}(l)) \mid l \in L,\ t \in \mathcal{T}\} \cup \{\mathsf{inv}_t(\mathit{write}(l, v)) \mid l \in L, v \in V, t \in \mathcal{T}\}$$
$$\mathcal{R}_{\mathrm{RW}} = \{\mathsf{resp}_t(v) \mid v \in V,\ t \in \mathcal{T}\} \cup \{\mathsf{resp}_t(\mathrm{ok}),\ t \in \mathcal{T}\}$$

We model the state of a read-write memory as a function *mem* from L to V. We say that a sequence *ops* of operations is *legal starting from mem* if it is a legal sequential history of read-write memory where *mem* is the initial state, and we denote the state resulting from applying *ops* by $mem/[wrSet]$, where *wrSet* is a partial function that maps each location written in *ops* to the last value written to it in *ops*. (We use $L \to V_\perp$ to denote the set of partial functions from L to V, where $\perp$ indicates where the function is not defined.)

Given a set $S \subseteq \mathcal{T}$, a *serialisation* of S is a sequence $\sigma \in S^*$ such that each transaction in S occurs exactly once in σ. We denote the set of serialisations of S by $ser(S)$. For a serialisation σ of S and transaction $t \in S$, we denote the prefix of σ up to and including t by $\sigma|_{\leq t}$. A serialisation is *consistent with* a partial order *po* on S if *po* is contained in the order imposed by the serialisation; we denote the set of such serialisations by $ser(S, po)$ Two partial orders on S are *consistent with each other* if there is a serialisation of S that is consistent with both. (We use "partial order" to refer to any transitive and antisymmetric, but not necessarily reflexive, binary relation.)

2.4 TM interfaces, correctness conditions, and models

We can model a TM system supporting object $\mathcal{O}$ using an automaton $TM(\mathcal{O})$, with $external(TM(\mathcal{O})) = \mathcal{I}_{TM(\mathcal{O})} \cup \mathcal{R}_{TM(\mathcal{O})}$, where:

$$\mathcal{I}_{TM(\mathcal{O})} = \mathcal{I}_{\mathcal{O}} \cup \{\mathsf{begin}_t, \mathsf{commit}_t, \mathsf{cancel}_t\}$$
$$\mathcal{R}_{TM(\mathcal{O})} = \mathcal{R}_{\mathcal{O}} \cup \{\mathsf{beginOk}_t, \mathsf{commitOk}_t, \mathsf{aborted}_t\}$$

These actions, together with actions of $internal(TM(\mathcal{O}))$, determine the behaviour of a TM system modeled this way. In Sections 3 and 5, we use such models to *specify* two notions of correct behaviour for a TM system. In our ongoing work, we also use such automata to model TM implementations. We can then use simulation proofs to prove that one correctness condition implements another (see Section 6 for a sketch of one such proof), or that a TM implementation implements a condition.

In our proof work, we model the automata in formal detail using the PVS language, and use the PVS theorem prover to construct and check our simulation proofs. However, for clarity, we present the automata using more familiar mathematical notation, and sketch the above-mentioned proof carefully but informally.

3 *WRC*: a weak correctness condition for TM

In this section, we present the *WRC* correctness condition for TM; we relate it to previous conditions in the next section. We define *WRC* as the set of traces exhibited by an I/O automaton. Expressing our conditions this way serves both to make them unambiguous, and to facilitate the construction of precise, machine-checked proofs about them, an important aspect of our work. *WRC* is defined by the automaton shown in Figure 1. The automaton should be mostly self-explanatory; we discuss the most interesting and important details below.

Roughly, *WRC* requires that, for every execution, there is a total order over committed transactions that respects their real-time order and the sequential semantics of the underlying objects. Thus, an implementation that satisfies *WRC* gives the appearance that each committed transaction takes effect atomically at some point during its execution. *WRC* further requires that a transaction that aborts observes behaviour consistent with an execution in which the (partially executed) transaction takes effect atomically at some point after it begins; this execution must be consistent with the real-time order of committed transactions. *WRC* allows:

- any transaction to "pretend" that some commit-pending transactions commit even though they may ultimately abort (a transaction is *commit-pending* if it has invoked commit, but has not yet committed or aborted);
- operations executed by any transaction to be "justified" by different sets of other transactions at different times during its execution; and
- transactions that ultimately abort to see any set of committed transactions in any order that *could have* existed at some point during its execution, even if this is not consistent with the transactions that actually commit in the execution.

The flexibility implied by this last point is acceptable because, as stated earlier, the key requirement for an aborted transaction is that it does not observe behaviour that *could not have occurred* during the interval of its execution: the user program should correctly handle any behaviour that *could* occur, so it does not matter if such a transaction observes behaviour that did not actually occur, because aborted transactions have no observable side effects.

The automaton achieves this permissive condition by requiring each successful transaction to be justified by a sequence of transactions that includes itself and respects the sequential semantics and the real-time order on transactions. The latter requirement is enforced as follows: First, the automaton records which committed transactions precede each transaction (see the *extOrder* variable and the begin_t action). Second, the sequence of transactions that justifies a transaction committing successfully must include all of the committed transactions that precede it in the real-time order, and the transaction itself (see the $\mathsf{commitOk}_t$ action and the *validCommit* predicate). It may also include any subset of the commit-pending transactions. The ability to include any subset of commit-pending transactions provides much of the flexibility of the *WRC* condition.

It may seem that, when a transaction t_2 commits successfully having chosen an order in which a commit-pending transaction t_1 precedes t_2, this requires t_1 to commit successfully and to be ordered before t_2. In fact, this is not the case. Rather, the automaton requires that, no matter what happens in the future, the successful commit of t_2 will always be justified. This *may* be by having t_1 commit successfully, but it could also be that t_2's successful commit is justified by another transaction t_3 by the time t_1 fails. This is illustrated by the following example. In this and all other example executions in the paper, we assume a read/write memory whose values are all zero initially. Apart from commit operations, all invocations are followed immediately by a corresponding response. B_1 denotes $\mathsf{begin}_1/\mathsf{beginOk}_1$, C_1 denotes commit_1, OK_1 denotes $\mathsf{commitOk}_1$, A_1 denotes $\mathsf{aborted}_1$, $\mathsf{W}_1\mathrm{x}1$ denotes $\mathsf{inv}_1(\mathit{write}(x,1))/\mathsf{resp}_1(\mathrm{ok})$, and $\mathsf{R}_1\mathrm{x}1$ denotes $\mathsf{inv}_1(\mathit{read}(x))/\mathsf{resp}_1(1)$.

$\mathsf{B}_1\ \mathsf{W}_1\mathrm{x}1\ \mathsf{C}_1\ \mathsf{B}_2\ \mathsf{R}_2\mathrm{x}1\ \mathsf{B}_3\ \mathsf{W}_3\mathrm{x}1\ \mathsf{C}_2\ \mathsf{C}_3\ \mathsf{A}_1\ \mathsf{OK}_2\ \mathsf{OK}_3$

State variables

$extOrder$: binary relation on $\mathcal{T}$; initially empty
For each $t \in \mathcal{T}$:
$status_t$: {$\mathsf{notStarted}, \mathsf{beginPending}, \mathsf{active}, \mathsf{opPending}, \mathsf{commitPending}, \mathsf{cancelPending}, \mathsf{committed}, \mathsf{aborted}$}; initially $\mathsf{notStarted}$
ops_t: sequence of operations (i.e., an element of $(\mathcal{I} \times \mathcal{R})^*$); initially empty
$pendingOp_t$: $\mathcal{I}$; initially arbitrary
$snapshots_t$: set of subsets of $\mathcal{T}$; initially empty

Actions

begin_t
Pre: $status_t = \mathsf{notStarted}$
Eff: $status_t \leftarrow \mathsf{beginPending}$
$extOrder \leftarrow extOrder \cup (CT \times \{t\})$

$\mathsf{inv}_t(op)$
Pre: $status_t = \mathsf{active}$
Eff: $status_t \leftarrow \mathsf{opPending}$
$pendingOp_t \leftarrow op$

commit_t
Pre: $status_t = \mathsf{active}$
Eff: $status_t \leftarrow \mathsf{commitPending}$

cancel_t
Pre: $status_t = \mathsf{active}$
Eff: $status_t \leftarrow \mathsf{cancelPending}$

snap_t
Pre: $status_t \in \{\mathsf{beginPending}, \mathsf{active}, \mathsf{opPending}\}$
Eff: $snapshots_t \leftarrow snapshots_t \cup \{CPT \cup CT\}$

$\mathsf{beginOk}_t$
Pre: $status_t = \mathsf{beginPending}$
Eff: $status_t \leftarrow \mathsf{active}$

$\mathsf{resp}_t(r)$
Pre: $status_t = \mathsf{opPending}$
$validResp(t, pendingOp_t, r)$
Eff: $status_t \leftarrow \mathsf{active}$
$ops_t \leftarrow ops_t \circ (pendingOp_t, r)$

$\mathsf{commitOk}_t$
Pre: $status_t = \mathsf{commitPending}$
$validCommit(t)$
Eff: $status_t \leftarrow \mathsf{committed}$

$\mathsf{aborted}_t$
Pre: $status_t \in \{\mathsf{beginPending}, \mathsf{opPending}, \mathsf{commitPending}, \mathsf{cancelPending}\}$
$status_t = \mathsf{commitPending} \implies validFail(t)$
Eff: $status_t \leftarrow \mathsf{aborted}$

Derived state variables, functions and predicates

$$ops(\sigma) \triangleq ops_{\sigma_0} \circ ops_{\sigma_1} \circ \ldots \circ ops_{\sigma_n} \text{ where } \sigma \text{ is a sequence of transactions, and } n = |\sigma| - 1$$
$$CT \triangleq \{t \mid status_t \in \{\mathsf{committed}\}\}$$
$$CPT \triangleq \{t \mid status_t \in \{\mathsf{commitPending}\}\}$$
$$validCommit(t) \triangleq \exists S \subseteq CPT, \exists \sigma \in ser(CT \cup S, extOrder), t \in S \text{ and } ops(\sigma) \text{ is a legal sequential history}$$
$$validFail(t) \triangleq \exists S \subseteq CPT, \exists \sigma \in ser(CT \cup S, extOrder), t \notin S \text{ and } ops(\sigma) \text{ is a legal sequential history}$$
$$validResp(t, op, r) \triangleq \exists S \in snapshots_t, \exists \sigma \in ser(S \cup \{t\}, extOrder), ops(\sigma|_{\leq t}) \circ (op, r) \text{ is a sequential legal history}$$

Fig. 1. The automaton used to define the *WRC* correctness condition for TM.

This observation leads to a rather surprising requirement for *aborting* transactions to have a nontrivial validation condition; in most models, any attempt to commit a transaction may fail. This validation condition (see $\mathsf{aborted}_t$ action and *validFail* predicate) is identical to the one for successful commit (discussed above), except that the failing transaction is *not* included in the order. In other words, before failing, a transaction must confirm that all of the transactions that have already decided to commit can be justified without committing that transaction.

Next, we explain how the automaton ensures that the sequence of operations executed by a transaction is always consistent with an execution that *could* have happened. The key idea is essentially the same as the way we validate that a transaction can commit safely. The key difference is that we wish to allow for the operations of a transaction to be considered by that transaction to be applied at *any* time during its execution. This provides additional flexibility over the commit validation condition, because it allows the transaction to "go back in time" to any point during its execution and pretend that some concurrent transactions that were previously commit-pending but have now committed had not committed.

To facilitate this, the internal snap_t action, which is enabled at any time after a transaction begins and before its commit_t action, takes a snapshot of the set of transactions that are either committed or commit-pending when the action occurs. This allows the transaction to use the point at which such a snapshot was taken to justify its sequence of operations so far. Note that, just as the successful committing of a transaction may be justified by different sequences of transactions at different times (explained above), different snapshots may be used to justify the operations of a transaction at different times.

Finally, we observe that the sequence of transactions that justifies successfully committing a transaction imposes no constraints on transactions that abort. Furthermore, the sequences of transactions that justify the operations executed by an uncommitted transaction, as well as aborting a transaction, impose no constraints on *other* incomplete or aborted transactions. Thus, they need not agree on which transactions commit or in which order. As we will see in the next section, this is the key difference between *WRC* and the previously proposed opacity condition.

4 Previous correctness conditions

In this section, we discuss two correctness conditions—*opacity* [8] and *virtual worlds consistency* (VWC) [10]—that have been proposed previously for similar contexts and with similar motivation. That is, these proposals suggest correctness conditions for TM runtime interfaces which ensure that even transactions that ultimately abort observe only consistent memory states before they abort. Both proposals illustrate important points about what behaviours should and should not be allowed to be exhibited by a TM implementation, but both have shortcomings too.

We were working towards our *WRC* condition to overcome what we saw as shortcomings of opacity (see below) before we became aware of VWC. At first we thought that our condition would be strictly weaker than both opacity and VWC. However, as it turns out, *WRC* excludes behaviours exhibited by both opacity and VWC, while capturing the best features of both. We explain below.

4.1 Opacity

Opacity [8] is the first attempt to formally define a correctness condition that requires all transactions (even aborting ones) to observe consistent behaviour, as *WRC* does. Opacity is expressed in terms of similar interface and assumptions as we use in this paper, although we use an explicit begin_t action for transactions, which is not used in [8]. This appears to be a mostly cosmetic difference.

The most substantive difference between *WRC* and opacity is that *WRC* allows different aborted transactions to disagree on which other transactions commit, and in what order. Thus, unless a programmer violates rules (which are often enforced by the compiler) prohibiting "leaking" information out of aborted transactions, opacity precludes implementations that programmers could not distinguish from one that satisfies opacity.

Finally, an interesting subtlety of opacity is that it is not prefix-closed. For example, a read executed by one transaction may not be justified until later when another transaction writes the value it read. In this case, the prefix of the execution up to the read does not satisfy opacity, whereas later, when the write has occurred in a transaction that has invoked its commit operation, it can.

The authors of [8] address this concern by imposing an additional requirement that, at all times, the execution produced so far by the implementation satisfies opacity. By specifying our correctness condition as the set of traces that can be exhibited by an I/O automaton, we ensure *a priori* that the condition is prefix-closed (because an automaton cannot produce an execution without first producing all of its prefixes). Furthermore our approach facilitates the construction of hierarchical proofs that are sufficiently precise to be machine-checked.

4.2 Virtual Worlds Consistency

Both VWC and *WRC* relax opacity's requirement to justify all committed and aborted transactions using a single order, simply requiring that aborted ones never observe behaviour that could not have occurred. This point is illustrated by the following example: B_1 R_1x0 B_2 W_2x2 C_2 OK_2 B_3 R_3x2 R_3y0 W_1y1 C_1 OK_1

This execution is not opaque because the only order for the committed transactions is $t_1 t_2$, while t_3 observes that t_2 has committed but not t_1. The execution does satisfy *WRC*, because the operations of t_3 are consistent with behaviour that *could* have occurred, namely t_1 might have aborted. VWC similarly allows aborted transactions to disagree with the set of committed transactions and/or their order.

We agree that opacity's requirements on aborted transactions are too strong, but as we explain below, we believe that VWC relaxes them too much. Therefore, *WRC* excludes some behaviours that VWC allows.

The interfaces and assumptions used to define VWC differ in several ways from those of *WRC* and opacity, which leads to significant differences in what conditions can be expressed, and what executions are allowed. VWC is defined only for the limited case of a read-write memory in which no two writes ever write the same value. Furthermore, unlike the interfaces of opacity and *WRC*, VWC does not model transaction commit as an interval with an invocation and a response, which prevents it from allowing the flexibility of opacity and of *WRC* to allow executions in which a read is justified by a write of a transaction that ultimately does not commit. Finally, VWC allows aborted transactions to ignore committed transactions that precede them in real time. This appears to derive from an assumption that transactions are the only means of communication, which we do not assume.

In summary, *WRC* includes the best features of both opacity and VWC, while excluding executions allowed by each of them. It is interesting to note, moreover, that *WRC* allows executions that are allowed by neither opacity nor VWC, as shown by the following example:

B_1 R_1x0 B_2 W_2x2 C_2 B_3 OK_2 B_4 R_4x2 R_4y0 R_3x0 W_3z3 C_3 R_4z3 W_1y1 C_1 OK_1 A_3 C_4 A_4

In this example, the only valid order of committed transactions is $t_1 t_2$, so the only states are $(\mathsf{x}, \mathsf{y}, \mathsf{z}) = (0, 0, 0)$, $(0, 1, 0)$, and $(2, 1, 0)$. But t_4 sees $(2, 0, 3)$ so the execution is not opaque. Furthermore, because no committed transaction writes 3 to z, this also shows that this execution is not VWC. However, all reads of all aborted transactions can be justified under *WRC*. The only nontrivial one is t_4's read of z. t_4 can take a snapshot right after C_3 that includes t_1, t_2, and t_3. The following order satisfies *validResp* using that snapshot: $t_2 t_3 t_4 t_1$ (note that the serialisation is required to be valid only up to and including t_4). That order justifies t_4 seeing $(2, 0, 3)$.

5 *PRAG*: a stricter, more pragmatic condition

In this section, we introduce the *PRAG* automaton (Figure 2), which defines a stricter TM condition than *WRC* that is closer to the intuition about the structure of many TM implementations. In Section 6, we sketch a proof that every trace of *PRAG* is also a trace of *WRC*. Thus, to prove that an implementation satisfies the *WRC* condition, it suffices to prove that it satisfies the *PRAG* condition.

Like most TM implementations, *PRAG* supports a read-write memory, not the general object semantics supported by *WRC*. In most TM implementations, for each successful writing transaction, there is a distinct point between its commit invocation and its response at which the transaction takes effect. However, some algorithms—such as TL2 [4]—allow read-only transactions to take effect at some point before they invoke commit. Furthermore, an active transaction must always have a point during its execution at which the memory contains the values observed thus far by the transaction. We explain how these properties are captured by the *PRAG* automaton below. (We ignore for now the history variables of *PRAG*, which are used only for the proof in the next section that *PRAG* implements *WRC*.)

The *PRAG* automaton records the sequence of states produced by successful writing transactions, adding a new state at the end of the sequence whenever such a transaction commits successfully; see the $\mathsf{doCommitWriter}_t$ action. (The sequence is indexed from 0, so that $states_0$ is the initial state of the memory, and $states_k$ is the state written by the kth writing transaction; we use $states_{\text{last}}$ to denote the last element of *states*). The reads of a successful writing transaction must be consistent with the last state in the sequence before its state is appended (see the $\mathsf{doCommitWriter}_t$ action and the *readCons* predicate). Because the state is always appended during the commit interval of a transaction (see the commit_t, $\mathsf{doCommitWriter}_t$, and $\mathsf{commitOk}_t$ actions), successful writing transactions can be ordered in a sequence that is consistent with the real-time order of transactions and with the sequential semantics.

It remains to describe how *PRAG* allows a transaction to justify its reads by any state that existed during its execution, including committing a read-only transaction. Observe that the automaton records the length of the sequence of states when a transaction begins (see the begin_t action). Reads executed during a transaction (see the doRead_t action), as well as entire read-only transactions (see the $\mathsf{doCommitReadOnly}_t$ action), must be justified by that state or a subsequent state,

State variables

$states$: sequence of functions mapping L to V; initially a singleton mapping all locations to 0

For each $t \in \mathcal{T}$:

- pc_t: PCvals; initially $\mathsf{notStarted}$
- $beginIdx_t$: $\mathbb{N}$; initially arbitrary
- $rdSet_t$: $L \to V_\perp$; initially all $\perp$
- $wrSet_t$: $L \to V_\perp$; initially all $\perp$

Auxiliary history variables

$extOrder$: binary relation on $\mathcal{T}$; initially empty

For each $t \in \mathcal{T}$:

- ops_t: sequence of operations (i.e., $(\mathcal{I} \times \mathcal{R})^*$); initially empty
- $pendingOp_t$: $\mathcal{I}$; initially arbitrary
- $commitIdx_t$: $\mathbb{N}$; initially arbitrary
- $snapshots_t$: set of subsets of $\mathcal{T}$; initially empty

Actions

begin_t
Pre: $pc_t = \mathsf{notStarted}$
Eff: $pc_t \leftarrow \mathsf{beginPending}$
$beginIdx_t \leftarrow |states| - 1$
$extOrder \leftarrow extOrder \cup (CT \times \{t\})$

$\mathsf{inv}_t(read(l))$
Pre: $pc_t = \mathsf{active}$
Eff: $pc_t \leftarrow \mathsf{doRead}(l)$
$pendingOp_t \leftarrow read(l)$

$\mathsf{inv}_t(write(l, v))$
Pre: $pc_t = \mathsf{active}$
Eff: $pc_t \leftarrow \mathsf{doWrite}(l, v)$
$pendingOp_t \leftarrow write(l, v)$

commit_t
Pre: $pc_t = \mathsf{active}$
Eff: if $\mathbf{dom}(wrSet_t)$ is empty then
  $pc_t \leftarrow \mathsf{doCommitReadOnly}$
else
  $pc_t \leftarrow \mathsf{doCommitWriter}$

cancel_t
Pre: $pc_t = \mathsf{active}$
Eff: $pc_t \leftarrow \mathsf{cancelPending}$

$\mathsf{doCommitReadOnly}_t(n)$
Pre: $pc_t = \mathsf{doCommitReadOnly}$
$validIdx(t, n)$
Eff: $pc_t \leftarrow \mathsf{commitRespOk}$
$commitIdx_t \leftarrow n$

$\mathsf{doRead}_t(l, n)$
Pre: $pc_t = \mathsf{doRead}(l)$
$l \in \mathbf{dom}(wrSet_t) \vee validIdx(t, n)$
Eff: if $l \in \mathbf{dom}(wrSet_t)$ then
  $pc_t \leftarrow \mathsf{readResp}(wrSet_t(l))$
else
  $v \leftarrow states_n(l)$
  $pc_t \leftarrow \mathsf{readResp}(v)$
  $rdSet_t \leftarrow rdSet_t/[l \to v]$

$\mathsf{beginOk}_t$
Pre: $pc_t = \mathsf{beginPending}$
Eff: $pc_t \leftarrow \mathsf{active}$

$\mathsf{resp}_t(v)$
Pre: $pc_t = \mathsf{readResp}(v)$
Eff: $pc_t \leftarrow \mathsf{active}$
$ops_t \leftarrow ops_t \circ (read(l), v)$
$snapshots_t \leftarrow snapshots_t \cup \{CPT \cup CT\}$

$\mathsf{resp}_t(\mathrm{ok})$
Pre: $pc_t = \mathsf{writeRespOk}$
Eff: $pc_t \leftarrow \mathsf{active}$
$ops_t \leftarrow ops_t \circ (write(l, v), \mathrm{ok})$
$snapshots_t \leftarrow snapshots_t \cup \{CPT \cup CT\}$

$\mathsf{commitOk}_t$
Pre: $pc_t = \mathsf{commitRespOk}$
Eff: $pc_t \leftarrow \mathsf{committed}$

$\mathsf{aborted}_t$
Pre: $pc_t \notin \{\mathsf{notStarted}, \mathsf{active}, \mathsf{commitRespOk}, \mathsf{committed}, \mathsf{aborted}\}$
Eff: $pc_t \leftarrow \mathsf{aborted}$

$\mathsf{doCommitWriter}_t$
Pre: $pc_t = \mathsf{doCommitWriter}$
$readCons(states_{\mathrm{last}}, rdSet_t)$
Eff: $pc_t \leftarrow \mathsf{commitRespOk}$
$commitIdx_t \leftarrow |states|$
$states \leftarrow states \circ states_{\mathrm{last}}/[wrSet_t]$

$\mathsf{doWrite}_t(l, v)$
Pre: $pc_t = \mathsf{doWrite}(l, v)$
Eff: $pc_t \leftarrow \mathsf{writeRespOk}$
$wrSet_t \leftarrow wrSet_t/[l \to v]$

Functions and predicates

$$readCons(mem, rdSet) \triangleq \forall l \in \mathbf{dom}(rdSet), rdSet_t(l) = mem(l)$$
$$validIdx(t, n) \triangleq beginIdx_t \leq n < |states| \wedge readCons(states_n, rdSet_t)$$

Fig. 2. The automaton used to define the *PRAG* correctness condition for TM. We use PCvals as shorthand for the set of all values assigned to *pc* variables.

that is, a state that existed during the execution of the transaction.

PRAG is similar to opacity in that it disallows many executions that would be acceptable for TM in order to achieve a simpler condition that is likely to suffice for a large class of practical implementations. To more precisely characterise the relationship between the two conditions (again, modulo cosmetic interface differences), we first observe that opacity allows some executions that *PRAG* does not. This is illustrated by the following example: B_1 R_1x0 W_1x1 C_1 B_2 OK_1 R_2x0 W_2y1 C_2 OK_2

Because t_2 reads 0 from x, t_2 must be ordered before t_1, which opacity allows. However, the $PRAG$ automaton must commit t_1 before ok_1, and therefore before t_2 invokes commit. Thus, when t_2 executes its doCommitWriter action, its validation will fail, as its read set is not consistent with the last state installed by t_1. Thus, $PRAG$ does not allow this execution.

We believe that every execution allowed by $PRAG$ is also allowed by opacity, but we have not formally proved this. It may be interesting or useful to express opacity (restricted to executions whose prefixes all satisfy opacity, of course) as an automaton, and to formally prove these relationships. However, we are more interested in identifying conditions that are useful in practice than in precise characterisations of relationships to previous conditions.

6 *PRAG* implements *WRC*

Here we sketch a proof that $PRAG$ implements WRC; we are working on constructing this proof formally using the PVS theorem prover system. We first describe the key ideas in the proof, and then present formal invariants that support it.

Recall that WRC imposes two conditions on an execution:

- There is a serialisation of all the committed transactions and some subset of commit-pending transactions that is consistent with their real-time order, such that applying the transactions according to that serialisation results in a legal sequential history.
- For any transaction t (including active and aborted transactions), there is a serialisation of t, together with all committed and some commit-pending transactions at some point during t's execution, that is consistent with real-time order such that applying the transactions according to that serialisation *up to* t results in a legal sequential history.

WRC does not require the serialisations of the second condition to be consistent with each other or with the serialisation of the first condition—indeed, even which commit-pending transactions are included in the serialisation may vary from transaction to transaction. However, $PRAG$ is more restrictive: a transaction can see values written by another transaction only if the other transaction is *effectively committed*, that is, it has executed its doCommit action. We denote the set of such transactions by

$$ECT = \{t \mid pc_t \in \{\mathsf{commitRespOk}, \mathsf{committed}\}\}$$

Furthermore, $PRAG$ guarantees that, at any time, there is a serialisation of all transactions that is consistent with their real-time order, and that satisfies both the first condition when restricted to transactions in ECT, and the second condition for any transaction t when restricted to transactions in $ECT \cup \{t\}$.

We use a simple kind of forward simulation proof in which, for each state of $PRAG$, there is a single state of WRC that is related to it by the simulation relation. That is, the simulation relation is a *refinement mapping*. To facilitate this, we

f maps states of $PRAG$ to states of WRC such that for any state s of $PRAG$,

$$f(s).extOrder = s.extOrder$$

and for all $t \in \mathcal{T}$,

$$f(s).status_t = \begin{cases} \mathsf{opPending} & \text{if } s.pc_t \in \{\mathsf{doRead}(l) \mid l \in L\} \cup \{\mathsf{readResp}(v) \mid v \in V\} \\ & \qquad \cup \{\mathsf{doWrite}(l,v) \mid l \in L \wedge v \in V\} \cup \{\mathsf{writeRespOk}\} \\ \mathsf{commitPending} & \text{if } s.pc_t \in \{\mathsf{doCommitReadOnly}, \mathsf{doCommitWriter}, \mathsf{commitRespOk}\} \\ s.pc_t & \text{otherwise} \end{cases}$$

$$f(s).ops_t = s.ops_t$$
$$f(s).pendingOp_t = s.pendingOp_t$$
$$f(s).snapshots_t = s.snapshots_t$$

Fig. 3. A refinement mapping from $PRAG$ to WRC.

augment $PRAG$ with history variables (shown in Figure 2): The *extOrder*, ops_t, $pendingOp_t$, $commitIdx_t$ and $snapshots_t$ variables simply maintain the corresponding state variables from WRC. The only nontrivial aspect to adding these variables is determining when to update $snapshots_t$, which is explained below. The refinement mapping from $PRAG$ to WRC is shown in Figure 3.

We now describe the proof. The correspondence between initial states is immediate. For the inductive part of the proof, the choice of WRC action(s) for a given action of $PRAG$ is fairly straightforward. For an internal action, the prestate and the poststate of $PRAG$ map to the same state of WRC, so no step is taken by WRC. For external actions other than $\mathsf{resp}_t(v)$ and $\mathsf{resp}_t(\mathrm{ok})$, we take the same action in WRC. For $\mathsf{resp}_t(v)$ and $\mathsf{resp}_t(\mathrm{ok})$, we choose a snap_t action followed by the resp_t action itself. (Although $PRAG$ does not exploit the flexibility afforded by having a separate snap_t action, we need a snapshot to satisfy the preconditions of the resp_t action, so we take one immediately before such actions.)

In most cases, the justification for the choice of action(s) follows directly from the prestate implied by the refinement mapping. However, for resp, commitOk and aborted, we must show that the appropriate validation condition holds. To facilitate this, we add the $commitIdx_t$ history variable; proving properties about this variable entails most of the complexity of the proof.

For $t \in ECT$, the $commitIdx_t$ variable maintains an index into *states* indicating a state that t either wrote (if t is a writing transaction) or against which it validated its read set (if it is read-only) in its doCommit action. Thus, that state can be thought of as the state of the memory immediately after the transaction. These "commit indices" define a partial order on effectively committed transactions that is consistent with *extOrder* (see Invariant 6). Furthermore, there is a bijection between states in *states* after the initial state and effectively committed writing transactions (see Invariant 5), so this order is total on these transactions.

Two key observations are: a) applying the effectively committed transactions according to any serialisation consistent with the commit-index order yields a legal sequential history (see Invariant 11); and b) every transaction t that has started but not effectively committed can be inserted into this serialisation at some point consistent with *extOrder*, such that applying the transactions in the prefix of this serialisation ending in t is also a legal sequential history (see Invariant 12).

That the validation conditions hold for $\mathsf{commitOk}_t$ and $\mathsf{aborted}_t$ follows from observation a) above. Because the commit-index order is consistent with *extOrder*, there is some serialisation that is consistent with both orders. Furthermore, because a transaction executing the commitOk action is in ECT, and one executing the aborted action is not in ECT, such a serialisation satisfies the appropriate validation condition for each of these actions.

Finally, we must also verify that the validation condition of the resp_t action is satisfied. This follows from observation b) above, using the snapshot just taken by the previous action (which includes all transactions in ECT).

We now present invariants of $PRAG$ that are used in the proof. The following ones are easy to verify by induction:

Invariant 1 *If* $pc_t = \mathsf{notStarted}$ *then* $rdSet_t(l) = wrSet_t(l) = \perp$ *for all* $l \in L$.

Invariant 2 *For* $t \in \mathcal{T}$*:*

- *If* $pc_t \neq \mathsf{notStarted}$ *then* $beginIdx_t < |states|$.
- *If* $t \in ECT$ *then* $beginIdx_t \leq commitIdx_t < |states|$.

Invariant 3 *For* $t \in \mathcal{T}$*:*

- *If* $pc_t = \mathsf{doCommitReadOnly}$ *then* $\mathbf{dom}(wrSet_t) = \emptyset$.
- *If* $pc_t = \mathsf{doCommitWriter}$ *then* $\mathbf{dom}(wrSet_t) \neq \emptyset$.

Invariant 4 says that if two transactions are ordered by *extOrder*, then the first transaction is effectively committed, the second transaction has started, and the begin index of the second transaction is no earlier than the commit index of the first. Furthermore, if the second transaction is effectively committed, then its commit index is no earlier than the commit index of the first, and is strictly later if the second transaction is a writing transaction.

Invariant 4 *If* $(t, t') \in extOrder$*, then:*

- $t \in ECT$
- $pc_{t'} \neq \mathsf{notStarted}$
- $commitIdx_t \leq beginIdx_{t'}$
- $t' \in ECT \implies commitIdx_t \leq commitIdx_{t'}$
- $t' \in ECT \wedge \mathbf{dom}(wrSet_{t'}) \neq \emptyset \implies commitIdx_t < commitIdx_{t'}$

Invariant 5 says that for every state in *states* except for the initial state, there is a unique transaction that writes the memory to produce that state. It is the first transaction that has that commit index.

Invariant 5 *If* $0 < n < |states|$ *then, for exactly one* $t \in ECT$*,* $commitIdx_t = n$ *and* $\mathbf{dom}(wrSet_t) \neq \emptyset$.

We denote the partial order on ECT defined by the commit indices by

$$\begin{aligned}commitIdxOrder = \{(t,t') \mid & commitIdx_t < commitIdx_{t'} \\ & \vee (commitIdx_t = commitIdx_{t'} \wedge \mathbf{dom}(wrSet_t) \neq \emptyset)\}\end{aligned}$$

Note that a writing transaction is ordered before any other transaction with the same commit index. This is a partial order on ECT because of Invariant 5, and it follows from Invariant 4 that this order is consistent with $extOrder$.

Invariant 6 *commitIdxOrder is consistent with extOrder.*

We now state several invariants about the operations done by transactions. To do so, we define the sequence of operations done by transaction t:

$$ops'_t = \begin{cases} ops_t \circ (pendingOp_t, v) & \text{if } pc_t = \mathsf{readResp}(v) \\ ops_t \circ (pendingOp_t, \mathrm{ok}) & \text{if } pc_t = \mathsf{writeRespOk} \\ ops_t & \text{otherwise} \end{cases}$$

Invariant 7 says that for every transaction, there is some state that was current at some time after the transaction began and is consistent with the transaction's read set. This holds because whenever a location is added to a read set (i.e., in one case of the doRead action), it is associated with the value of the location in some state that is already consistent with the transaction's read set.

Invariant 7 *If $pc_t \neq \mathsf{notStarted}$ then $validIdx(t,n)$ for some n.*

Invariant 8 says that a location is in the write set of a transaction t if and only if t has written that location, in which case, it stores the last value so written. This holds because whenever a transaction writes a location (doWrite action), it adds the location to its write set, associating it with the value written.

Invariant 8 *$l \in \mathbf{dom}(wrSet_t)$ if and only if ops'_t contains $(write(l,v), ok)$ for some $v \in V$. Furthermore, if there is such an operation in ops'_t then the last one has $v = wrSet_t(l)$.*

Invariant 9 says that applying the operations of a transaction starting from any state that is consistent with the transaction's read set yields the appropriate responses, and leaves the memory so that the locations in the transaction's write set have the values specified by the write set and all other locations are unchanged. It follows straightforwardly by induction because a *read* operation gets the last value written to that location (by Invariant 8), or the value in the state from which the transaction starts if the location is not written by the transaction.

Invariant 9 *If $readCons(mem, rdSet_t)$ then ops'_t is legal starting from mem and the state resulting from applying ops'_t is $mem/[wrSet_t]$.*

We now come to the key invariants used to prove that *PRAG* implements *WRC*. Invariant 10 says that for any effectively committed transaction $t \in ECT$, applying the operations of the effectively committed transactions up to and including t in any order consistent with the commit-index order yields the appropriate responses and leaves the memory in the state corresponding to the commit index of t. This invariant follows by induction, using Invariant 9 to show that it is preserved by doCommit actions. Invariants 11 and 12 make observations a) and b) above precise. Invariant 11 is just a special case of Invariant 10, while Invariant 12 follows from Invariants 9 and 10.

Invariant 10 *If* $\sigma \in ser(ECT, commitIdxOrder)$ *then for every* $t \in ECT$*,* $ops(\sigma|_{\leq t})$ *is a legal sequential history, and applying it to the initial state of the memory leaves the memory in state* $states_n$*, where* $n = commitIdx_t$*.*

Invariant 11 *If* $\sigma \in ser(ECT, commitIdxOrder)$ *then* $ops(\sigma)$ *is a legal sequential history.*

Invariant 12 *If* $pc_t \neq \mathsf{notStarted}$*,* $validIdx(t, n)$*,* $S = \{t' \in ECT \mid commitIdx_{t'} \leq n\}$ *and* $\sigma \in ser(S, commitIdxOrder)$ *then* $ops(\sigma) \circ ops'_t$ *is a legal sequential history.*

7 Ongoing and future work

We have introduced a general TM correctness condition *WRC*, and a more restrictive but more intuitive condition *PRAG*. We are working on constructing formal, machine-checked proofs that *PRAG* implements *WRC*, as sketched in the previous section, and that some TM implementations implement *PRAG*.

We are starting with a simple version of the popular TL2 TM algorithm [4]. This simple version, called *TL2-CG*, uses coarser-grained synchronisation than is consistent with current multiprocessor architectures, but allows us to illustrate some key ideas in our approach. Future work includes refining *TL2-CG* to successively more realistic implementations, ultimately proving a realistic model of the TL2 implementation correct. We aim to make it easy to reuse parts of the proof, for example to prove variants on the algorithm, of which there are many.

One important aspect of our work is to eliminate the need to prove backward simulations [12], which are particularly challenging, and are necessary for many TM implementations. To this end, as in our previous work [3,5], we plan to identify one or more general *intermediate automata* that we can prove (using backward simulation proofs) implement *PRAG*. The idea is that we and others can then prove the correctness of a TM implementation by proving that it implements one of these intermediate automata using only forward simulation.

There are numerous aspects of TM models and implementations that our work does not yet address, including nontransactional memory accesses, nesting, privatisation and publication idioms, progress properties, and notions of dangerous vs. safe code, which relax consistency requirements for code that is known (or constructed) to be safe even if the transaction executing it observes inconsistent behaviour.

References

[1] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *CAV '08: Proceedings of the 20th International Conference on Computer Aided Verification*, pages 121–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying concurrent data structures by simulation. In Eerke Boiten and John Derrick, editors, *Proceedings of the RefineNet Workshop 2005 (REFINE 2005), Guildford, UK*, Electronic Notes in Theoretical Computer Science, April 2005.

[3] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV '06)*, 2006.

[4] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *International Symposium on Distributed Computing*, pages 194–208, 2006.

[5] Simon Doherty and Mark Moir. Nonblocking algorithms and backward simulation. In *Proceedings of 23rd Annual Symposium on Distributed Computing (DISC)*, 2009.

[6] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Completeness and nondeterminism in model checking transactional memories. In *CONCUR '08: Proceedings of the 19th international conference on Concurrency Theory*, pages 21–35, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 321–336, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Rachid Guerraoui and Michal Kapałka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[10] Damien Imbs, José Demendívil, and Michel Raynal. On the consistency conditions of transactional memories. Technical Report 1917, Institut de Recherche en Informatique et Systèmes Aalátoires, 2008.

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

[12] Nancy Lynch and Frits Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[13] The PVS Specification and Verification System, http://pvs.csl.sri.com/.