A Scalable and Oblivious Atomicity Assertion

Rachid Guerraoui and Marko Vukolić

School of Computer and Communication Sciences, EPFL,
INR, Station 14, CH-1015 Lausanne, Switzerland
{rachid.guerraoui, marko.vukolic}@epfl.ch

Technical Report LPD-REPORT-2008-011
June 9, 2008

Abstract. This paper presents SOAR: the first oblivious atomicity assertion with polynomial complexity. SOAR enables to check atomicity of a single-writer multi-reader register implementation. The basic idea underlying the low overhead induced by SOAR lies in greedily checking, in a backward manner, specific points of an execution where register operations could be linearized, rather than exploring all possible precedence relations among these.
We illustrate the use of SOAR by implementing it in +CAL. The performance of the resulting automatic verification outperforms comparable approaches by more than an order of magnitude already in executions with only 6 read/write operations. This difference increases to 3-4 orders of magnitude in the "negative" scenario, i.e., when checking some non-atomic execution, with only 5 operations. For example, checking atomicity of every possible execution of a single-writer single-reader (SWSR) register with at most 2 write and 3 read operations with the state of the art oblivious assertion takes more than 58 hours to complete, whereas SOAR takes just 9 seconds.

1 Introduction

With multi-core architectures becoming mainstream, concurrent programming is expected to become the norm, even among average developers who might not always have the right skills and experience. Concurrent programming is however notoriously difficult. In particular, it is hard to control the interference between concurrent threads without compromising correctness on the one hand, or restricting parallelism on the other hand.

Among consistency criteria for concurrent programming, *atomicity* (also known as *linearizability* [14]) is one of the most popular. This is because atomicity reduces the difficult problem of reasoning about a concurrent program into the simpler problem of reasoning about its sequential counterpart. Roughly speaking, atomicity guarantees that concurrently-executing requests on shared objects appear sequential: namely, each request appears to be executed at some point (known as the *linearization point* [14]) between its invocation and response time (real-time ordering). An example of an atomic execution of a read/write register

is depicted in Figure 1, along with its linearization points (assuming the register is initialized to 0). In contrast, the execution in Figure 2 is not atomic. This is because we cannot place linearization points such that the sequential specification of a register is satisfied, i.e., every read returns the last value written.

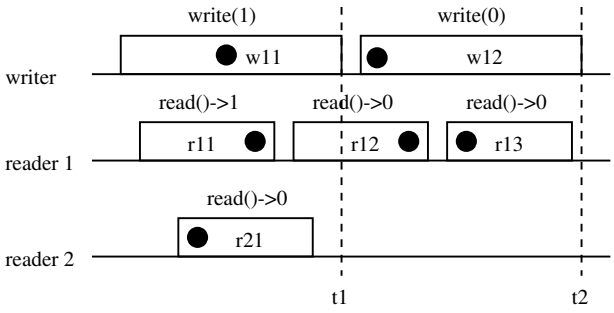

Fig. 1. Example of an atomic execution.

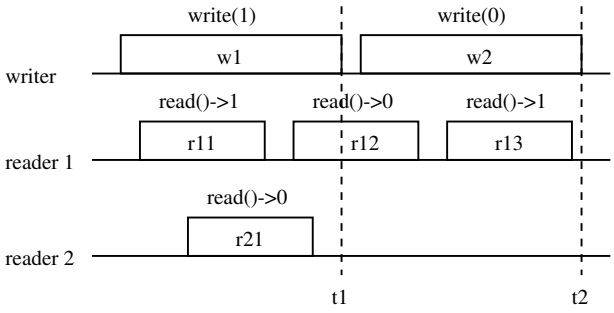

Fig. 2. Example of a non-atomic execution.

Precisely because it simplifies the job of the programmers by encapsulating the difficulty underlying synchronizing shared atomic objects, atomicity is hard to implement. As pointed out in [7], an evidence of this difficulty is that several published implementations of atomic shared memory objects have later shown to be incorrect. Not surprisingly, tools for checking atomicity are of crucial importance, in particular automatic ones that are suitable for machine verification [11].

So far, tools for checking atomicity have mainly been designed for specific programming languages (e.g., Concurrent Java [10]). Some exceptions have been proposed in the form of *language-oblivious* execution assertions, which enable to check the atomicity of implementation histories. Some of these (e.g., [15,16]) are

still *non-algorithm-oblivious* in the sense that a fair amount of knowledge about the checked algorithm is needed in order to check correctness.

Genuinely oblivious assertions were proposed in [24] (Lemma 13.16) and [18]. These assertions do not require any knowledge, neither about the language nor about the checked algorithm. One specific such assertion is of particular interest: the one of Chockler et al. (Property 1 of [7]) for it was especially devised for automatic verification. This assertion, which we refer to as CLMT, can be written as a simple logical predicate and is very appealing for automatic verification especially when paired with a model checker such as the TLC for the +CAL algorithm language [22, 23].

Unfortunately, the CLMT assertion does not scale well as we discuss below. Consider the (non-atomic) execution on a single-writer multi-reader (SWMR) read/write register depicted in Figure 2. When implemented in +CAL, the CLMT assertion takes more than one minute on our 4 dual-core Opteron machine to verify that this execution is not-atomic. This is even without taking into the account the operations invoked by $reader2$. When considering a single operation of $reader2$, the verification takes hours.

On the other hand, it is very simple for a human to verify manually that the execution of Figure 2 is not-atomic. For the execution to be atomic, the linearization point of the write operation $w1$ must come before that of read $r11$, since $r11$ does not return the initial value 0. Similarly, $w2$ must be linearized before $r12$. This leaves $r13$ which violates the sequential specification of the read/write register, meaning that the execution is not atomic.

What makes CLMT slow is the very fact that it reasons about atomicity by identifying the adequate properties of a *precedence relation* among read/write operations. Namely, CLMT checks atomicity by establishing the existence of a precedence relation among operations that: a) is a non-reflexive partial order, and b) satisfies certain (five different) properties. Without diving into the details of these properties, it is easy to see that this verification scheme cannot scale for it does imposes an *exponential* computational complexity on a model checker. Namely, with $2^{|op|^2}$ different possible relations over the set of $|op|$ different operations, there is simply too many relations to check, even for modest values of $|op|$, regardless of the nature of the properties that are to be checked. This is especially true when the "good" precedence relation does not exist, i.e., when the execution is not atomic. The motivation of this paper is to ask whether it is possible to devise an oblivious, yet scalable atomicity assertion.

We present *SOAR (Scalable and Oblivious Atomicity asseRtion)*, the first oblivious atomicity assertion with polynomial complexity. SOAR is devised for single-writer multi-reader concurrent objects, of which the single-writer multi-reader register is a very popular representative [6, 24]. Indeed, many applications of the register abstraction make use mainly of its single-writer variant. Such applications include for example consensus [2, 5, 12] as well as snapshot implementations [3].

Like CLMT, SOAR gives a sufficient condition for atomicity (in fact, in Section 3.3 we prove SOAR equivalent to CLMT in our single-writer setting).

Interestingly, we could also use SOAR in +CAL to verify that some seemingly natural simplifications of the celebrated Tromp's algorithm [26] (implementing an atomic bit out of three safe bits) lead to incorrect solutions. By doing this, we show that our SOAR implementation in +CAL can be used successfully in identifying non-atomic executions and algorithm debugging.

SOAR has a low-degree polynomial complexity ($O(|op|^3)$ in the worst case). It outperforms CLMT [7] by more than an order of magnitude already in verifying atomicity of executions with only 6 read/write operations.[1] This difference increases to 3-4 orders of magnitude in the "negative" scenario, i.e., when checking some non-atomic execution. For example, checking atomicity of every possible execution of a single-writer single-reader (SWSR) register with at most 2 write and 3 read operations with CLMT takes more than 58 hours to complete, whereas SOAR takes just 9 seconds.

Underlying SOAR lies the idea of *greedy linearization*. Basically, SOAR looks for linearization points in an execution ex rather than checks for precedence relations. SOAR performs its search in a backward manner starting from the end of the execution, linearizing the last write operation in ex (say w) and then trying to linearize as many read operations as possible *after* w. Then, the linearized operations are removed from ex and the linearization reiterates. It is important to emphasize that the greedy linearization is without loss of generality. We prove this by establishing the equivalence between SOAR and CLMT (for the single-writer case). As we pointed out however, SOAR is designed specifically for verifying atomicity of single writer objects, whereas CLMT is a general assertion suitable also for multi-writer applications.

While SOAR is specified with an atomic read/write data structure in mind, we believe that it is not difficult to extend it to cover other atomic objects in which only one process can change the state of the object (single-writer). Extending SOAR and the underlying greedy linearization idea to optimize model checking of multi-writer objects is very interesting open problem. This is left as future work.

The rest of the paper is organized as follows. After giving some preliminary definitions in Section 2, we describe our assertion in details and prove its correctness in Section 3. In Section 4 we illustrate how SOAR can be used for model checking Tromp's algorithm and its variations in +CAL/TLC. We also report on some performance measurements. We overview related work in Section 5.

2 Preliminaries

2.1 Processes and objects

We model processes and shared objects using the non-deterministic I/O Automata model [25]. We simply give here the elements that are needed to recall

[1] We always compare SOAR to a version of CLMT that is optimized for the single-writer case as we discuss in Section 4.2.

atomicity, state our assertion and prove its correctness. In short, an I/O automaton is a state machine whose state can change by discrete atomic transitions called *actions*. We consider two sets of processes: a singleton *writer* and a set of processes called *readers* (we refer to a process belonging to the union of these sets as *client*).

A read/write register is a shared object consisting of the following:

1. set of values D, with a special value v_0 (called the initial value),
2. set of operations $write(v)$, $v \in D$ and $read()$
3. set of responses $D \cup \{ack\}$,
4. sequential specification of the register is any sequence of read/write operations such that the responses of operations comply with the following:
   (a) $write(v) \triangleq x := v$; return ack (where x is initialized to v_0)
   (b) $read() \triangleq$ return x

To access the register, a client issues an *operation descriptor* that consists of the identifier of the operation $id \in \{`write', `read'\}$ and the identifier of the client; in case of a write, a value v is added to the descriptor. To simplify the presentation, we sometimes refer to an operation descriptor *op* simply as an operation *op*. A single-writer multi-reader (SWMR) register is a read/write object in which only the process *writer* may issue write operations. We denote by wrs (resp., rds) the set of write (resp., read) operations.

Clients use the actions of the form $\mathsf{invoke}(op)$ and $\mathsf{response}(op, v)$, where $op \in wrs \cup rds$ and $v \in D \cup \{ack\}$, to invoke operations and to receive responses. A sequence β of invoke and response actions is called an *execution*. An invoked operation *op* is said to be complete (in some execution β) if β contains $\mathsf{response}(op, v)$, for some $v \in D \cup \{ack\}$ (we say $\mathsf{response}(op, v)$ *matches* $\mathsf{invoke}(op)$). An operation *op* is said to be *pending* in β if β contains the $\mathsf{invoke}(op)$ action but not its matching response.

The execution *ex* is *sequential* if (a) the first action is an invocation, (b) each invocation, except possibly the last, is immediately followed by its matching response, and (c) every response is immediately followed by an invocation.

We say that an execution β is *well-formed* if (1) for every $\mathsf{response}(op, v)$ action in β there is a unique $\mathsf{invoke}(op)$ action in β that precedes $\mathsf{response}(op, v)$, (2) for every client c there is at most one pending operation issued by c in β.

Moreover, we assume that each well-formed execution β contains the invocation and the response action of the special operation $w_0 = write(v_0)$ called the *initial write*, such that the response action for w_0 precedes invocations of any other operation. All executions considered in this paper are assumed to be well-formed. A well-formed, sequential execution β is called *legal*, if β is in the sequential specification of the register.

Finally, we say that a complete operation *op precedes* an operation op' (or, alternatively, that op' *follows* *op*) in a well formed execution β if the response action of *op* precedes the invocation action of op' in β (we denote this by $op <_\beta op'$). Let *op* and op' be two invoked operations in β; if neither $op <_\beta op'$), nor $op' <_\beta op$), we say that *op* and op' are *concurrent* (in β).

2.2 Atomicity

We define atomicity (or linearizability) in the following way [6]: a (well-formed) execution β is atomic if there is a permutation $\pi(\beta)$ of all operations in ex such that: (1) $\pi(ex)$ is legal, and (2) if $op <_{\beta} op'$ then $op <_{\pi(\beta)} op'$.

In this paper we rely on the *Partial Order (PO)* property [7] for proving atomicity. As shown in Lemma 2 of [7], PO is sufficient for atomicity, i.e., if β satisfies PO then β is atomic. Moreover, we use the PO property to establish the correctness of our atomicity assertion in Section 3.3.

Definition 1. ***(PO Property).*** *Let op be the set of all operations invoked in the execution β that contains no pending operations and wrs (resp., rds) subset of all writes (resp., reads) in op. An execution β satisfies a* Partial Order *(PO) property if there is an irreflexive partial ordering $\prec$ on all elements of op, such that, in β:*

1. $\forall \pi, \phi \in op$, *if* $\pi <_{\beta} \phi$, *then* $\neg(\phi \prec \pi)$.
2. $\forall \pi, \phi \in wrs$, *either* $\pi \prec \phi$ *or* $\phi \prec \pi$.[2]
3. $\forall \pi \in wrs, \forall \phi \in rds$, *if* $\pi <_{\beta} \phi$, *then* $\pi \prec \phi$.
4. $\forall \pi, \phi \in rds$, *if* $\pi <_{\beta} \phi$ *then for each* $w \in LastPrecWrites(\pi, \prec)$, $w \prec \phi$.
5. *Let* $\pi \in rds$ *and let* v *be the value returned by* π. *Then,* v *is written by some write* $w \in LastPrecWrites(\pi, \prec)$.

Above, $LastPrecWrites(\pi, \prec) == \{w \in wrs : (w \prec \pi) \wedge \neg(\exists w' \in wrs : (w \prec w') \wedge (w' \prec \pi)\}$.

The PO property can be simply written as a logical predicate (assertion), to which we refer as CLMT.

3 A Scalable and Oblivious Atomicity asseRtion (SOAR)

3.1 Intuition: Greedy linearization

Our SOAR assertion is motivated by the observation that it is easy to linearize (in the single-writer case) the fragments of the execution between every two writes. Consider for example the fragment of the execution of Figure 2 in between initial time t_0 and time t_1, the time of completion of write $w1$, that contains only those read operations that are invoked before t_1 (i.e., r_{11}, r_{12} and r_{21}). It is clear that only read operations that return the value written by w_0 (say v_0) can be linearized between w_0 and w_1. Moreover, such reads cannot be preceded by reads that return values other than v_0. In other words, in the execution of Figure 2, only r_{21} can be linearized between w_0 and w_1 while the other reads must be linearized after w_1. We can repeat this partitioning of the execution between two writes and apply the above reasoning iteratively, until we exhaust all write

[2] In our case, with the single writer, this property becomes (having in mind Property 1) becomes: $\forall \pi, \phi \in wrs$, if $\pi <_{\beta} \phi$ then $\pi \prec \phi$.

operations. When a single write operation w_W is left, the remaining (still non-linearized) read operations must return the value written by w_W in order for the execution to be atomic. In the example of Figure 2 the operations would be linearized in the following order: w_0, r_{21}, w_1, r_{11}, w_2, r_{12}, leaving r_{13} which actually violates the sequential specification of the atomic read/write register.

The greedy linearization idea described above is based on checking the fragments of the execution that are between every two writes, starting from the beginning of the execution. While this is the natural way for a human to linearize executions, this approach leads to reasoning about execution suffixes (that remain after removing linearized operations). In our case, we found it more convenient to reason formally about execution prefixes; hence, we choose to apply greedy linearization starting from the end of the execution, using the similar idea. Consider, again the execution of Figure 2. It is trivial to see that the last write to be linearized is w_2. Now we can try to linearize as many reads as possible *after* w_2; however, this cannot be done with any of the reads. We can remove all linearized operations from the execution (i.e., in our case, only w_2) and apply the same reasoning to the remaining execution prefix. However, before reiterating, we must make sure that removing linearized operations indeed leaves us with the execution prefix; more concretely, we must check that none of the reads that will remain in the execution was invoked after the completion of the linearized write. In the case of w_2, this condition is satisfied (no operations are invoked after w_2 completes). In the next iteration, we would linearize w_1 and r_{13}. Finally, in the last iteration we could see that the atomicity is violated since not all of the remaining read operations return the value written by the initial write (r_{11} returns 1).

3.2 Description

We formalize our greedy linearization approach to obtain a generic assertion for atomicity in the following way. We denote:

- by W the total number of writes (not counting the initial write) in some execution ex that contains no incomplete operations,
- by w_i the i^{th} write in ex,
- by rds_W the set of all read operations in ex, and
- by $ex_i^{rds_i}$ $(i = 0 \dots W)$ the prefix of the execution ex that contains only write operations from w_0 to w_i, and only read operations from set rds_i.

Notice that $ex_W^{rds_W} \equiv ex$.

We assert the atomicity of every partial execution $ex_i^{rds_i}$ $(i = 0 \dots W)$ as follows:

1. If $i = 0$ (i.e., if $ex' = ex_0^{rds_0}$ contains only one write) then ex' is atomic if and only if all (read) operations from rds_0 return the initial value,
2. else (i.e., if $i > 0$), we:
   (a) remove from rds_i every read r that satisfies the following properties (we denote the set of such reads $linRds(i)$:

i. r returns the value written by the write w_i,
ii. r does not precede w_i, and
iii. if some $r' \in R_i$ follows r, then r' returns the value written by w_i.

Basically, the reads from the set $linRds(i)$ are immediately linearizable and SOAR greedily linearizes such reads.

(b) If there is a read in $rds_i \setminus linRds(i)$ that follows w_i, then $ex_i^{rds_i}$ is not atomic.

(c) $ex_i^{rds_i}$ is atomic if and only if $ex_{i-1}^{rds_{i-1}}$ is atomic, where $rds_{i-1} = rds_i \setminus linRds(i)$.

Given the recursive nature of SOAR, the assertion can be written more compactly (and more precisely) as a logical predicate (Figure 3). We write it as follows, using the TLA+ [21].

$linRds(wrs, rds, Inv, Resp, Ret) == \{r \in rds :$
  $\bigwedge\ Ret[r] = Ret[lastWR(wrs)]$ * l1
  $\bigwedge\ Resp[r] > Inv[lastWR(wrs)]$ * l2
  $\bigwedge\ \forall r' \in rds : Resp[r] < Inv[r'] \Rightarrow Ret[r'] = Ret[lastWR(wrs)]\}$ * l3

$SOAR(wrs, rds, Inv, Resp, Ret) ==$
  IF $wrs = \{lastWR(wrs)\}$
  THEN $\forall r \in rds : Ret[r] = Ret[lastWR(wrs)]$ * a1
  ELSE
    $\bigwedge\ \forall r \in rds \setminus linRds(wrs, rds, Inv, Resp, Ret) : \neg(Inv[r] > Resp[lastWR(wrs)])$ * a2
    $\bigwedge\ SOAR(wrs \setminus \{lastWR(wrs)\}, rds \setminus linRds(wrs, rds, Inv, Resp, Ret), Inv, Resp, Ret)$ * a3

Fig. 3. SOAR as a TLA+ predicate

In Figure 3, $SOAR()$ takes five arguments: (i) the sets wrs and rds containing the identifiers of all write and read operations in the execution ex, respectively, (ii) the functions (arrays) $Inv, Resp : wrs \cup rds \rightarrow Nat$ (where Nat is the set of natural numbers), containing the global logical time [17] of invocations and responses of operations, respectively, and (iii) the function (array) $Ret : wrs \cup rds \rightarrow D$ (where D is the domain of values that an implemented read/write register can assume), which maps the operations to values which are written/read. Moreover, SOAR makes use of the function $lastWR(wrs)$ which returns the write in wrs that follows all other writes in ex.[3]

It is not difficult to see that the very approach that underlies SOAR yields a low degree polynomial complexity ($O(|op|^3)$ in the worst case, where op is the number of operations in the execution), which is to be contrasted with the exponential one of the CLMT assertion.

3.3 Correctness

To establish the correctness of SOAR we rely on the CLMT assertion, defined by the PO property, Def. 1, Section 2.2. We prove the correctness of SOAR by

[3] For simplicity, we assume that the identifiers of write operations are monotonically increasing with the time of operation invocation. If this is not the case the $lastWR()$ function should also take the function $Inv()$ as the argument.

showing its equivalence with CLMT (in our single writer multi-reader model). First, in Lemma 1, we show that each sequence of read/write operation β (that does not contain incomplete operations) for which SOAR returns true, also satisfies the PO property. Then, in Lemma 2, we show that whenever β satisfies PO, SOAR returns true.

Lemma 1. *If the assertion SOAR of Figure 3 applied on the sequence of read/write operations β returns TRUE, then β satisfies the PO property.*

Proof. Assume the assertion $SOAR()$ on β returns $TRUE$, and denote by wrs (resp., rds) the set of all write (resp., read) operations in β. Denote also $|wrs|-1$ by W (i.e., W is the number of non-initial writes in β). Consider the write operations w_i and subsets of read operations R_i defined as follows:

- for $i = 0 \ldots W$, $w_i = lastWR(wrs_i)$, where wrs_W=wrs and $wrs_{i-1} = wrs_i \setminus lastWR(wrs_i)$ $(i = 1..W)$, and
- for $i = 1 \ldots W$, $R_i = linRds(wrs_i, rds_i)$[4], where $rds_W = rds$, $rds_{i-1} = rds_i \setminus R_i$ and $R_0 = rds_0$.

In other words, each w_i is one of the W write operations in wrs, whereas each R_i is (a possibly empty) subset of rds. By construction of R_i and rds_i and Figure 3, it is not difficult to see that sets R_i are pairwise non-intersecting and that $\bigcup_i R_i = rds$.

Moreover, consider the following precedence relation $\prec$:

(*) $w_0 \prec R_0 \prec ... \prec w_i \prec R_i \prec ... \prec w_W \prec R_W$, where we implicitly think of $\prec$ as of transitive relation.

Notice that two elements (read operations) belonging to the same R_i are not ordered by the relation $\prec$. Obviously, $\prec$ is an irreflexive partial order and, moreover, if $\pi \prec \phi$ then $\neg(\phi \prec \pi)$ (for any two operations π and ϕ). We now prove that the relation $\prec$, as defined by (*), satisfies each of the 5 properties of Definition 1.

1. To prove Property 1, we distinguish 4 cases:
   (a) $\pi, \phi \in wrs$. Property 1 is satisfied by the implementation of the function $lastWR()$,
   (b) $\pi \in wrs$, $\phi \in rds$. Fix i, such that $\pi = w_i$. If $i > 0$, suppose by contradiction that $\phi \in R_j$ such that $j < i$. Then, $\phi \in rds_j$, i.e., $\phi \in rds_i \setminus linRds(wrs_i, rds_i)$. Since π completes before ϕ is invoked, by condition in line $a2$ of Figure 3, SOAR returns FALSE — a contradiction. On the other hand, if $\pi = w_0$, then $\neg\phi \prec \pi$ by (*).
   (c) $\pi \in rds$, $\phi \in wrs$. Fix $i > 0$ such that $\phi = w_i$. Suppose by contradiction that $\pi \in R_j = linRds(wrs_j, rds_j)$, such that $j > i$. Since π precedes w_i which precedes w_j, π precedes w_j (i.e., π completes before w_j is invoked). Hence, by line $l2$ of Figure 3, $\pi \notin R_j$ — a contradiction.

[4] In this proof, for simplicity of presentation, arguments 3-5 for $linRds()$ are implicitly assumed.

(d) $\pi, \phi \in rds$. Fix i, j such that $\pi \in R_i$ and $\phi \in R_j$. Suppose by contradiction that $j < i$. In this case, $\pi, \phi \in rds_i$. If π and ϕ return the same value, then, since $\pi \in R_i$, by lines $l1 - l3$ of Figure 3, $\phi \in R_i$, and $j = i$ — a contradiction. In case π and ϕ do not return the same value, then, by line $l3$ of Figure 3, $\pi \notin R_i$ — a contradiction.

2. Property 2 trivially follows from the definition of $\prec$ (see (*)).
3. To prove Property 3, note that all pairs of writes and reads are related by $\prec$ (see (*)), and by the Proof of Property 1 (case (b)) $\neg(\phi \prec \pi)$. Hence, $\pi \prec \phi$.
4. To prove Property 4, fix i such that $\pi \in R_i$. Observe that, by (*), $LastPrecWrites(\pi, \prec) = \{w_i\}$. By the Proof of Property 1 (case(d)), it is possible that either $\phi \in R_i$, or $\pi \prec \phi$. In both cases, by (*), $w_i \prec \phi$.
5. To prove Property 5, again, we fix i such that $\pi \in R_i$. By (*), $LastPrecWrites(\pi, \prec) = \{w_i\}$. Suppose, by contradiction, that π does not return the value written by w_i. If $i > 0$, by line $l1$ of Figure 3, $\pi \notin R_i$ — a contradiction. Similarly, if $i = 0$, $\pi \notin R_i$ by line $a1$ of Figure 3 — a contradiction. □

Lemma 2. *If the sequence of read/write operations β satisfies the PO property, then the assertion SOAR of Figure 3 applied on β returns $TRUE$.*

Proof. We prove this lemma by induction on the number of write operations in β. In the following, we denote by W_β the number of (non-initial) write operations in β, and by wrs (resp., rds) the set of write (resp., $read$) operations in β.

Base step: ($W_\beta = 0$) In this case, the only write in β is the initial write w_0. By definition of w_0, in every execution, w_0 completes before any other operation is invoked. Since β satisfies PO, by Property 3 of PO, w_0 precedes all reads in β. Moreover, since w_0 is the only write in β, for any read π in β, $LastPrecWrites(\pi, \prec) = \{w_0\}$. Hence, all reads in β return v_0. By definition of $lastWR()$, we have $lastWR(wrs) = \{w_0\}$. Hence, by line $a1$ of Figure 3, $SOAR()$ returns $TRUE$.

Induction hypothesis: Suppose Lemma 2 holds for all $W_\beta < k$, for some natural number k.

Induction step:. We show that Lemma 2 holds for $W_\beta = k$. Since β satisfies PO, there is a irreflexive partial order relation $\prec$, and non empty sets of read/write operations $OP_1 \dots OP_m$ (where $m \leq k + 1 + |rds|$, such that $\bigcup_{i=1\dots m} OP_i = wrs \cup rds$ and, for all i, j, $\forall op_i \in OP_i, op_j \in OP_j : op_i \prec op_j \equiv i < j$. Moreover, by Property 2 of PO, there is no OP_i that contains two different write operations.

Denote by OP_{lwr} the set OP_i that contains a write operation, such that all sets OP_j (if any), where $i < j$, contain only read operations. We denote the union of all such sets OP_j by OP' and the write contained in OP_{lwr} by w_k. Note that, for all op' in OP' $w_k \prec op'$.

Consider SOAR applied to β. Since in case $k > 0$ $wrs \neq \{lastWR(wrs)\}$, in order to prove that SOAR returns $TRUE$, we prove that the conditions in lines $a2$ and $a3$, Figure 3 hold.

To prove that the condition in line $a2$ holds, we first prove that $\forall op' \in OP' : op' \in linRds(wrs, rds)$ (where arguments 3-5 of $linRds()$ are implicit).

- First, notice that, for all op' in OP', $LastPrecWrites(op', \prec) = \{w_k\}$. Hence, by Property 5 of PO, every op' returns the value written by w_k, i.e., the condition in line $l1$ of the predicate $linRds(wrs, rds)$ is satisfied by each $op' \in OP'$.
- Furthermore, by Property 1 of PO, since $w_k \prec op'$ for each $op' \in OP'$, it is not possible that some op' completes before w_k is invoked. Hence, the condition in line $l2$ of the predicate $linRds(wrs, rds)$ is satisfied by each $op' \in OP'$.
- Finally, by Property 4 of PO and since for each $op' \in OP'$ $LastPrecWrites(op', \prec) = \{w_k\}$, if op' completes before some read op'' is invoked then $w_k \prec op''$ — i.e., $op'' \in OP'$. Moreover, by Property 5 of PO, all (read) operations in OP' return the value written by w_k. Hence, the condition in line $l3$ of the predicate $linRds(wrs, rds)$ is satisfied by each $op' \in OP'$.

Since all operations in OP' are in $linRds(wrs, rds)$, it is not difficult to see that the condition in line $a2$ of Figure 3 holds. Indeed, if this condition is false then there is a read rd that follows w_k such that $\neg(w_k \prec rd)$, a violation of Property 3 of PO.

To prove that the condition in line $a3$ of Figure 3 holds, we first show that the execution β' obtained after removing read operations from $linRds(wrs, rds)$ and w_k satisfies PO, with the same relation $\prec$. Properties 1-3 of PO for β' follow directly from the corresponding Properties for β. To prove Properties 4 and 5, notice that for all operations op from β', $w_k \notin LastPrecWrites(op, \prec)$ (otherwise $w_k \prec op$ and op in OP' and $linRds(wrs, rds)$, i.e., op is not in β'). Notice also that β' has the same write operations as β, except for w_k. Hence, Properties 4 and 5 of PO hold in β' as well. Finally, β' satisfies PO and $W_{\beta'} = k - 1$ — hence, by the Induction hypothesis, the condition in line $a3$ evaluates to $TRUE$. □

4 Application to Tromp's algorithm

We applied SOAR and CLMT to the celebrated algorithm of Tromp [26] which we implemented in the +CAL algorithm language. We compared SOAR and CLMT performance, and evaluated SOAR's applicability to detection of non-atomic executions and, hence, to debugging.

Our +CAL implementation of Tromp's algorithm with SOAR is given in Figure 4[5] (the code used for testing is given in Figure 5). It consists of two parts: (1) the SOAR part (comprised of lines 006-011, 037-043, 058-060, 064-068 and 102-106), and (2) the +CAL implementation of the Tromp's algorithm (comprised of the remaining lines of Figure 4). We explain both parts of the code, starting with Part 2 (Tromp's algorithm). In the following we refer to Figure 4.

[5] In Figure 4, '043:' denotes a line number added for simplicity of presentation, whereas 'l17:' denotes a +CAL label.

```
001: ---------------------------- MODULE TrompSOAR2 ----------------------------
002: EXTENDS Naturals, TLC, Sequences
003: CONSTANT MAXWRITE, MAXREAD, V, W, R, WRITER, READER, SOAR(_,_,_,_,_)

004: (* –algorithm Tromp
005:   variables
006:     (* 1. History variables used by SOAR and CLMT. *)
007:     globalClock = 0, writeCount = 0,
008:     readCount = MAXWRITE,wrs = {0},rds = {},
009:     Ret = [i ∈ 0 . . . MAXWRITE + MAXREAD ↦ 0],
010:     Inv = [i ∈ 0 . . . MAXWRITE + MAXREAD ↦ 0],
011:     Resp = [i ∈ 0 . . . MAXWRITE + MAXREAD ↦ 0]

012:     (* 2. Variables used to simulate safe registers. *)
013:     busy = [i ∈ {V, W, R} ↦ FALSE],
014:     value = [i ∈ {V, W, R} ↦ 0],

015:     (* 3. Tromp's algorithm variables. *)
016:     oldValue = 0,   (* Used by the writer*)
017:     R_writer = 0,   (* Used by the writer to read R*)
018:     W_reader = 0,   (* Used by the reader to read W*)
019:     v = 0, x = 0, returnValue = 0 (* Used by the reader*)

020:   (* 4. Safe register simulation. *)
021:   macro RW_INIT(reg) begin
022:     if ⋁((reg ∈ V, W) ∧ (self = WRITER))
023:        ⋁((reg = R) ∧ (self = READER))
024:     then busy[reg] := TRUE;
025:     end if;
026:   end macro

027:   macro READ(reg, result) begin
028:     if busy[reg] = FALSE then result := value[reg];
029:     else either result := 0 or result := 1 end either;
030:     end if;
031:   end macro

032:   macro WRITE(reg, val) begin
033:     value[reg] := val; busy[reg] := FALSE;
034:   end macro

035:   (* 5. Tromp's Algorithm w. SOAR. *)
036:   procedure write(val) begin l1:
037:     (* Update of history variables*)
038:     writeCount := writeCount + 1;
039:     globalClock := globalClock + 1;
040:     Inv[writeCount] := globalClock;
041:     Resp[writeCount] := INF;
042:     Ret[writeCount] := val;
043:     wrs := wrs ∪ {writeCount};

044:     (* Tromp's algorithm write() code*)
045:       if oldValue ≠ val then
046:         (* change V *)
047: l2:      RW_INIT(V);
048: l3:      WRITE(V, val);
049:         oldValue := val;
050:         (* if W=R then change W *)
051: l4:      RW_INIT(R);
052: l5:      READ(R, R_writer);
053:         if value[W] = R_writer then
054: l6:        RW_INIT(W);
055: l7:        WRITE(W, 1 − value[W]);
056:         end if;
057:       end if;
058: l8:     (* Update of history variables and return*)
059:       globalClock := globalClock + 1;
060:       Resp[writeCount] := globalClock;
061:       return;
062:   end procedure

063:   procedure read() begin l9:
064:     (* Update of history variables *)
065:     globalClock := globalClock + 1;
066:     readCount := readCount + 1;
067:     rds := rds ∪ {readCount};
068:     Inv[readCount] := globalClock;

069:     (* Tromp's algorithm read() code*)
070:       (* If W=R return v - line 1 *)
071: l10:   RW_INIT(W);
072: l11:   READ(W,W_reader);
073:       if W_reader = value[R] then
074:         returnValue := v;
075:       else
076:         (* x : = Read(V) - line 2 *)
077: l12:     RW_INIT(V);
078: l13:     READ(V, x);
079:         (* If W≠R change R - line 3 *)
080: l14:     RW_INIT(W);
081: l15:     READ(W, W_reader);
082:         if W_reader ≠ value[R] then
083: l16:       RW_INIT(R);
084: l17:       READ(R, 1 − value[R]);
085:         end if;
086:         (* v : = Read(V) - line 4 *)
087: l18:     RW_INIT(V);
088: l19:     READ(V, v);
089:         (* If W=R return v - line 5 *)
090: l20:     RW_INIT(W);
091: l21:     READ(W, W_Reader);
092:         if W_reader = value[R] then
093:           returnValue := v;
094:         else
095:           (* v : = Read(V) - line 6 *)
096: l22:       RW_INIT(V);
097: l23:       READ(V, v);
098:           (* return x - line 7 *)
099:           returnValue := x;
100:         end if;
101:       end if;

102: l24:(* Update of history variables and return*)
103:     Ret[readCount] := returnValue;
104:     globalClock := globalClock + 1;
105:     Resp[readCount] := globalClock;
106:     assert (SOAR(wrs, rds, Inv, Resp, Ret));
107:     return;
108:   end procedure
```

Fig. 4. SOAR application to Tromp's algorithm

```
109:  (* 6. Code for testing. *)
110:  process Writer = WRITER begin wrloop:
111:    while (writeCount < MAXWRITE) ∧ (readCount ≤ MAXWRITE + MAXREAD) do
112:      either call write(0) or call write(1) end either;
113:    end while;
114:  end process;

115:  process Reader = READER begin rdloop:
116:    while (writeCount ≤ MAXWRITE) ∧ (readCount < MAXWRITE + MAXREAD) do
117:      call read();
118:    end while;
119:  end process;
120: end algorithm
*)
```

Fig. 5. SOAR application to Tromp's algorithm (continued)

In short, Tromp's algorithm gives an implementation of a single-writer single-reader (SWSR) atomic bit, using 3 safe[6] [18] (SWSR) bits: V, W and R, all initialized to 0. Bits V and W are owned (written) by the writer, whereas R is owned by the reader of the atomic bit. To simulate safe registers in +CAL, we use the variables *busy* and *value* (lines 012-014), as well as macros in lines 020-034. The main code of the Tromp's algorithm is given in lines 044-057 (the write code) and 069-101 (the read code). Comments in these portions of code (e.g., in lines 046, 050, or 070, 076, etc.) give the lines of the pseudocode as stated in the original paper [26]. Below each such comment, there is a +CAL translation of the corresponding pseudocode.

The SOAR part of the code in Figure 4 consists of operations on certain history variables necessary for the implementation of SOAR (as well as CLMT). History variables [1] play no role in the algorithm and serve only for the assertions. These lines are written as a wrapper around the code of the original algorithm in an *oblivious* manner; namely, no lines are inserted in the main code of Tromp's algorithm. For example, lines 037-043 and 058-060 are wrapped around the original write code, whereas lines 064-068 and 102-106 are wrapped around the original read code. Below, we explain in details the history variables required by SOAR.

First, SOAR requires history variables wrs and rds (sets of write and read operations), as well as history arrays (functions) $Inv[]$, $Resp[]$ and $Ret[]$. Initially, $wrs = \{0\}$, i.e., wrs contains the identifier of the initial write w_0, while $Inv[0] = Resp[0] = 0$ and $Ret[0] = v_0 = 0$ (lines 008-011). Besides, the following three history variables are also needed (line 007): (i) *globalClock* to act as a global clock and, hence, facilitate the implementation of functions $Inv[]$ and $Resp[]$, and (ii) *writeCount* and *readCount* the counters for write and read operation identifiers, respectively, which take values from non-overlapping domains. All these variables/arrays are accessed only at the beginning (invocation) and the end (completion) of read/write operations. We believe that the operations on

[6] Basically, a safe register ensures that a read rd returns the last value written only if rd is not concurrent with any write. In case of concurrency, a read may return an arbitrary value.

history variables are very intuitive and simple to follow. We clarify, however, two lines: (a) in line 041, the response time of the newly invoked write is set to INF, where INF (infinity) represents a constant that such that the *globalClock* cannot get greater than INF, and (b) in line 103, the returned value of the read is taken from the $returnValue$ variable in which the main read code of Tromp's algorithm (lines 069-101) stores the read value.

Finally, constants $MAXWRITE$ (resp., $MAXREAD$) denote the maximum number of write (resp., read) operations invoked in the checked execution.

4.1 Asserting non-atomic executions

We used our implementation of Figure 4 to verify that certain, seemingly plausible, "optimizations" of Tromp's algorithm lead to the incorrect solution.

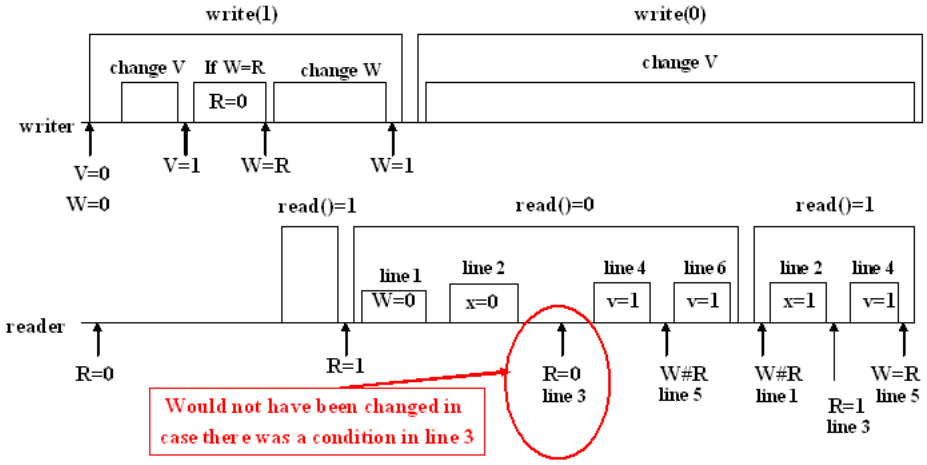

Fig. 6. Violation of atomicity after removing the condition in line 3 of Tromp's algorithm.

For example, it is not straightforward to see why the condition 'if $W \neq R$' in line 3 of the Tromp's read pseudocode is necessary (see line 079, Fig. 4) knowing that this line is executed only if indeed $W \neq R$ in line 1 of the original read pseudocode (line 070, Fig. 4). However, removing this condition (i.e., lines 080-082 and 085 of Fig. 4) leads to a violation of atomicity, which can be detected by SOAR. Using the error output of the TLC model checker, we were able to extract the execution that leads to the atomicity violation (see Figure 6). Interestingly, such "simplified" Tromp's algorithm remains regular [18], but it is not atomic. In a similar way, we were also able to show that the instruction in line 6 of the original pseudocode (line 095, Fig. 4) is also necessary. This demonstrates the usability of SOAR in debugging and asserting non-atomicity in practice.

4.2 Performance

All our performance results are obtained running TLC model checker (using 4 processors) on a 4 dual-core Opteron 8216 with 8 GB of RAM. TLC model checker is ran on an implementation of the Tromp's algorithm in +CAL, varying the number of invoked read/write operations.

Model checking was done to verify the atomicity of the Tromp's algorithm using both the CLMT and SOAR. Obtained graphs are given in Figure 7. Results are given for a specific variation of the CLMT, optimized for a single writer scenario. Notably the optimization modifies the condition 2 of Definition 1, Section 2.2 to impose that for any precedence relation $\prec$ and every $i \in 1 \dots W-1$ $w_i \prec w_{i+1}$ (where W is the total number of writes, represented by the variable *writeCount* in our +CAL implementation, Figure 4). Moreover, the initial write w_0 was always pre-linearized before running the CLMT assertion, which significantly improves its performance.

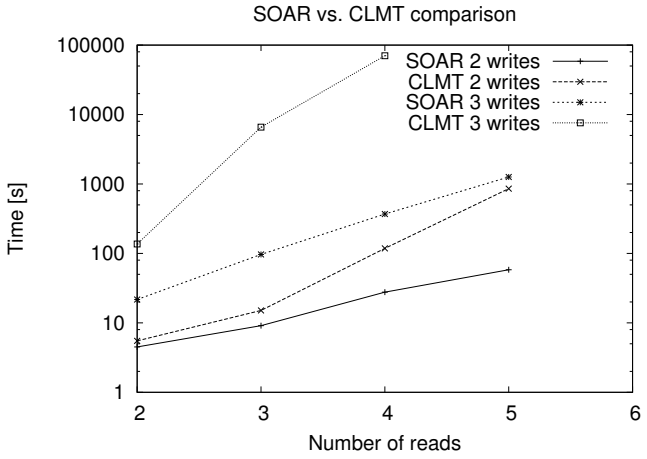

Fig. 7. SOAR vs. CLMT comparison. The entire time required for model checking Tromp's algorithm is represented.

From Figure 7 it can be seen that already in model checks of Tromp's algorithm with as few as 6 read/write operations (e.g., with 3 reads and 3 writes) a model check with SOAR takes more than an order of magnitude less time than with CLMT. The difference is even more glaring if a non-atomic execution is checked. For example, it takes only 15 milliseconds for SOAR to state that an execution of 2 (without the read r_{21}) is not atomic, whereas CLMT takes more than 70 seconds. This represents a difference of 3-4 orders of magnitude already for an execution with only 5 operations, and, by its design, the complexity of CLMT grows exponentially with the number of operations in the execution.

In practice, when checking executions with a fairly small number of operations, SOAR is as fast as any assertion maintaining the global clock can be. By maintaining the global clock, we mean maintaining the execution history in the form of: 1) set of all operations invoked in the execution, 2) arrays of operations' invocation and response times, and 3) the array of values written/read by operations. Indeed, our results show that, for all the points represented in Figure 7, SOAR introduces no visible overhead with respect to a dummy assertion that maintains the global clock.

5 Concluding Remarks

The concept of an *atomic* object was first introduced by Lamport [19, 20] in the context of read/write registers. This concept was later extended to objects other than registers by Herlihy and Wing [14], under the notion of *linearizability*. In this paper, we use notions of atomicity and linearizability interchangeably.

Atomicity assertions were proposed by Hesselink [15, 16]. These assertions are not *oblivious* since they are based on the history variables that are inserted in specific places of the checked algorithm. A fair amount of knowledge of the checked algorithm is thus required.

As we discussed in the introduction, Chockler et al., [7] proposed a genuinely *oblivious* atomicity assertion (quoted CLMT) that does not require any knowledge, neither on the language nor on the algorithm. In [7], CLMT has been used as the basis for the Partial Order machine automaton, that was in turn used in forward simulations to prove the correctness of various atomic object implementations (another simulation based atomicity proof (of a lock-free queue) can be found in the paper by Doherty et al. [8]). However, as we show in this paper, CLMT imposes exponential complexity on the model checker. This is not surprising given the result of Alur et al. [4], showing that model checking linearizability is in EXPSPACE. SOAR circumvents this result by focusing on the single-writer implementations.

In [26], Tromp proposed an atomicity automaton suitable for designing and verifying atomic variable constructions. The automaton nodes represent the state of a run on the atomic variable, whereas transitions represent read and write operations. This automaton addresses only the single-writer single-reader atomic constructions.

Some work was also devoted to checking the atomicity of transactional blocks of code, e.g., [9, 10, 13].

The simple greedy linearization idea that we employ in this paper is not new. A similar idea was exploited by Wang and Stoller [27] as one of the steps in the context of atomicity inference for programs with non-blocking synchronization.

Acknowledgments

We thank Gregory Chockler, Eli Gafni and Leslie Lamport for interesting discussions and very useful comments.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.
2. Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.
3. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.
4. Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.
5. James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.
6. Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.
7. Gregory Chockler, Nancy Lynch, Sayan Mitra, and Joshua Tauber. Proving atomicity: An assertional approach. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 152–168, September 2005.
8. Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In David de Frutos-Escrig and Manuel Nez, editors, *FORTE*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2004.
9. Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.
10. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM.
11. Cormac Flanagan and Shaz Qadeer. Atomicity for reliable concurrent software. In *A tutorial at the ACM SIGPLAN 2005 conference on Programming language design and implementation (PLDI'05)*, 2005.
12. Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.
13. Rachid Guerraoui, Thomas Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *PLDI '08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation*, 2008.
14. Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
15. Wim H. Hesselink. An assertional criterion for atomicity. *Acta Informatica*, 38(5):343–366, 2002.
16. Wim H. Hesselink. A criterion for atomicity revisited. *Acta Informatica*, 44(2):123–151, 2007.
17. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
18. Leslie Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, May 1986.

19. Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.
20. Leslie Lamport. On interprocess communication. part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.
21. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
22. Leslie Lamport. The +CAL algorithm language. In *NCA '06: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, page 5, Washington, DC, USA, 2006. IEEE Computer Society.
23. Leslie Lamport. Checking a multithreaded algorithm with +CAL. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 151–163, September 2006.
24. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1996.
25. Nancy A. Lynch and Mark R.Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
26. John Tromp. How to construct an atomic variable (extended abstract). In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 292–302, London, UK, 1989. Springer-Verlag.
27. Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 61–71, New York, NY, USA, 2005. ACM.