Information and Computation **160**, 167–188 (2000)
doi:10.1006/inco.1999.2847, available online at http://www.idealibrary.com on IDEAL®

Model-Checking of Correctness Conditions for Concurrent Objects[1]

Rajeev Alur

Bell Laboratories, P.O. Box 636, 600 Mountain Avenue, Murray Hill, New Jersey 07974-0636 and University of Pennsylvania, Department of Computer and Information Science, 200 South 33rd Street, Philadelphia, Pennsylvania 19003
E-mail: alur@cis.upenn.edu

Ken McMillan

Cadence Berkeley Laboratories, 2001 Addison Street, Third Floor, Berkeley, California 94720
E-mail: mcmillan@cadence.com

and

Doron Peled

Bell Laboratories, P.O. Box 636, 600 Mountain Avenue, Murray Hill, New Jersey 07974-636 and The Technion—Israel Institute of Technology, Technion City, Haifa 32000, Israel
E-mail: doron@bell-labs.com

The notions of serializability, linearizability, and sequential consistency are used in the specification of concurrent systems. We show that the model checking problem for each of these properties can be cast in terms of the containment of one regular language in another regular language shuffled using a semicommutative alphabet. The three model checking problems are shown to be, respectively, in PSPACE, in EXPSPACE, and undecidable. © 2000 Academic Press

1. INTRODUCTION

A common way of specifying concurrent systems is to describe the desired sequential behavior of the system and then to allow the implementation to execute certain operations in parallel, provided the appearance of sequential behavior is maintained for a suitable observer. The earliest such notion of correctness was *serializability* (see, for instance, [EGLT76, Pap86, BHG87]), which requires that a collection of transactions that are scheduled in parallel must produce the same

[1] A preliminary version of this paper appeared in *Proceedingss of the 11th IEEE Symposium on Logic in Computer Science* (LICS 1996), pp. 219–228.

result as the same transactions scheduled in some sequential order. Thus, an observer without the knowledge of the actual order of scheduling would not be able to infer that the transactions were not executed sequentially. A more abstract notion of correctness of a concurrent implementation is *sequential consistency* [Lam79]. In this case, an abstract specification of the desired sequential behavior is provided, and the concurrent implementation is required to produce behaviors that appear correct to an observer that has knowledge of only the local history of each parallel process. The notion of linearizability [HW90] is similar, but an observer knows, apart from local histories, also the ordering between any two transactions of different processes that do not overlap in time.

Each of these notions of correctness has its place. There are cases when serializability is adequate (as in database applications). In other cases, such as cache coherence, an abstract service specification is required, and hence sequential consistency is the appropriate correctness criterion (although it is sometimes relaxed in practice). In still other cases, especially the implementation of concurrent objects in software, the stricter requirement of linearizability is met. It ensures that when the client's invocation to some operation on a concurrent object has returned, the effects of the operation have been committed and will be visible to all future calls by other clients. This allows clients without pending calls to communicate with each other without shattering the illusion of sequentiality.

Implementations of such specifications are often based on fairly subtle protocols between concurrent processes. While the correctness of many of the standard solutions (e.g., two-phase locking for serializability) has been proved rigorously using proof theory (see, for instance, [LMWF94]), the specific implementations are still prone to bugs due to the optimizations introduced by the designers. Because of indeterminacy of scheduling and communication latency, they are subject to complex race conditions and deadlocks that can easily go undetected in testing and simulation due to their infrequency of occurrence. Thus, it is desirable to formally verify that the protocol meets its specification in all circumstances. The technique of model checking suggests itself for this purpose, since the protocols involved can in many cases be effectively modeled as finite state machines, at least with enough generality to examine the concurrency issues involved. This raises the question of the complexity of verifying concurrency properties on finite state models.

The complexity of deciding sequential consistency and serializability for a single finite execution trace has been previously studied (we will call this the *membership problem*). For the case of serializability[2], this membership problem has a polynomial algorithm [EGLT76]. The serializability problem for regular languages has also been treated in the context of trace theory [FR85]; however complexity results were not obtained. For sequential consistency the membership problem is known to be NP-complete [GK92], and for linearizability it is also NP-complete [GK92], though it is in P if the number of processes is bounded. The complexity of the model checking problem in these two cases has not been studied, to our knowledge.

[2] Our notion of serializability has also been referred to as conflict-serializability [Pap86]. There is a weaker notion called view-serializability [Pap86], for which the membership problem is NP-complete. View serializability, however, does not fit into the general class of properties studied in this paper.

In this paper, we show that each of the three model checking problems—serializability, sequential consistency, and linearizability—can be cast in terms of the containment of one regular language in a regular language on a semi-commutative alphabet. The ability to commute alphabet symbols corresponds to the observer's inability to distinguish the order of occurrence of certain concurrent events. Our results are that for serializability the model checking problem is in PSPACE, for linearizability it is in EXPSPACE, and for sequential consistency it is undecidable.

2. PROBLEM DEFINITIONS

2.1. Preliminaries

Language Operations

For a string σ over an alphabet Σ and a subset Σ' of Σ, the *projection* of σ to Σ', denoted $\sigma \uparrow \Sigma'$, is the string obtained by deleting symbols not in Σ'. Let L_j be a language over an alphabet Σ_j for $j = 1 \ldots n$. The *asynchronous product* $\|_j L_j$ is the language L over the alphabet $\bigcup_j \Sigma_j$ such that a string σ is in L iff for each j, $\sigma \uparrow \Sigma_j$ is in L_j.

Traces

A *concurrent alphabet* is a pair (Σ, D), where Σ is a finite alphabet and D is a binary relation over Σ called the *dependency relation*. Unlike in trace theory [Maz87], we do not require D to be symmetric. Two symbols a and b are *commutable* (or independent) iff $(a, b) \notin D$. For a concurrent alphabet (Σ, D), define $\Rightarrow_D$ to be the least binary relation over Σ^* satisfying

(1) $\Rightarrow_D$ is reflexive and transitive, and

(2) for all strings $\sigma, \sigma' \in \Sigma^*$ and $(a, b) \notin D$, $\sigma \cdot ab \cdot \sigma' \Rightarrow_D \sigma \cdot ba \cdot \sigma'$.

Thus, $\sigma \Rightarrow_D \sigma'$ precisely when the string σ' can be obtained from σ by repeatedly commuting commutable pairs of symbols. Given a concurrent alphabet (Σ, D) and language L over Σ, the *closure* of L with respect to D, denoted $cl_D(L)$, consists of all strings σ' such that $\sigma' \Rightarrow_D \sigma$ for some $\sigma \in L$.

EXAMPLE. Let $\Sigma = \{a, b\}$ and $L = (ab)^*$. For $D = \{(b, a)\}$, $cl_D(L)$ contains all strings σ such that σ contains the same number of a's and b's, and in every prefix of σ, the number of b's does not exceed the number of a's. On the other hand, for $D = \emptyset$, $cl_D(L)$ contains all strings with the same number of a's and b's.

Note that when D is symmetric, the relation $\Rightarrow_D$ is also symmetric and hence an equivalence relation over Σ^*. Then, the equivalence classes of $\Rightarrow_D$ are called *Mazurkiewicz traces*.

The next lemma establishes that the closure operation preserves context-sensitivity of a language.

LEMMA 1. *If (Σ, D) is a concurrent alphabet, and L is a context-sensitive language over Σ, then $cl_D(L)$ is context-sensitive.*

Proof. One can modify a given context-sensitive grammar for L as follows: introduce a new nonterminal A for each terminal symbol a. Then, replace each occurrence of a in the rules by A. For each terminal symbol a and the corresponding new nonterminal A, add the rewriting rule $A \rightarrow a$. Finally, for each pair of symbols $(a, b) \notin D$ and corresponding new nonterminals A and B, add a rewriting rule $AB \rightarrow BA$. ▌

Unfortunately, the closure operation does not preserve regularity. The next example shows that it is possible that L is regular but $cl_D(L)$ is not even context-free.

EXAMPLE. Let $\Sigma = \{a, b, c\}$, $L = (abc)^*$, and $D = \emptyset$. The closure $cl_D(L)$ contains all the strings with the same number of a's, b's and c's. This is known to be context-sensitive and not context-free.

For all the correctness conditions that we consider, the verification problem can be reduced to checking language-inclusion $L \subseteq cl_D(L')$ for suitably chosen D and regular languages L and L'.

Specification of Objects

The definition of an *object* (or an abstract data type) A consists of a signature and a specification. The *signature* of an object A consists of a finite set $O(A)$ of operations and for every operation $o \in O(A)$, a set W_o of input values for the operation o and a set V_o of responses that the operation o may return. Let P be a finite set of processes. The event $o(p, A, w, v)$, for an operation $o \in O(A)$, a process $p \in P$, and values $w \in W_o$ and $v \in V_o$, denotes the event that the object A returns the response v when the process p applies the operation o with argument w. The alphabet $\Sigma(A)$ consists of all events of A. Each object also has a specification that tells which sequences of operations are legal. A *specification* $S(A)$ of an object A is a language over the alphabet $\Sigma(A)$.[3]

EXAMPLE. An *atomic* bit has two operations read and write. The read operation has no argument and returns either 0 or 1. The input to the write operation can be either 0 or 1, and it returns no value. For the sake of concise notation, we will drop the unused argument or value components of operations and use labels such as *read*(0) to denote the disjunction $+_p$ *read*$(p, x, 0)$, where the register name x is understood. The specification of the atomic bit is the language of the automaton shown in Fig. 1.

[3] Typical specifications have additional properties. Usually the language $S(A)$ is prefix-closed, deterministic, input-enabled, and symmetric with respect to process names; that is, for every string σ in $S(A)$, for every operation o and argument w, there is a unique response v such that for every process p, σ extended with $o(p, A, w, v)$ is in $S(A)$. For our purpose of complexity bounds, we do not impose these restrictions.

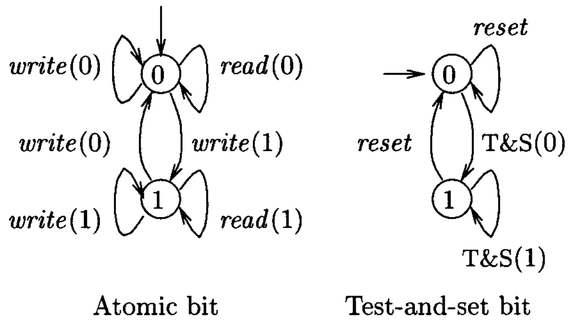

FIG. 1. Specification of Test-and-set and atomic bits.

Example. A *test-and-set* bit has two operations T&S and reset. The T&S operation has no argument and may return either 0 or 1. The reset operation has no argument and does not return any value. The specification of *test-and-set* bit is the language of the automaton of Fig. 1.

The object A is said to be finite-state with size k if k is a bound on the number of operations, the number of possible input arguments, the number of possible output responses, and the number of states of the NFA generating $S(A)$. Thus, both the objects of Fig. 1 are finite-state with size 2.

2.2. Sequential Consistency

The intuition behind sequential consistency (introduced by Lamport [Lam79]) is that an implementation of a collection of concurrent objects should appear to be correct to an observer that is able to record the history of each individual process, but has no global clock by which to determine the relative order of events of different processes.

Example. In the case of the atomic bit x, the event sequence

$$\mathit{read}(p, x, 0), \qquad \mathit{write}(p', x, 1)$$

meets the object's specification (recall that 0 is the initial value). On the other hand, the event sequence

$$\mathit{write}(p', x, 1), \qquad \mathit{read}(p, x, 0)$$

does not meet the specification. It is sequentially consistent, however, since the histories of the two individual processes are the same as those of the correct sequence. The sequence

$$\mathit{read}(p, x, 1), \qquad \mathit{write}(p', x, 0)$$

neither meets the specification nor is sequentially consistent, since every correct sequence in which p reads 1 must contain an operation that writes 1 to x.

Let Σ be the set of events of all objects. The specification S is the asynchronous product $\|_A S(A)$; that is, a string meets the sequential specification if its projections on individual objects satisfy their respective specifications. We say that a string σ is *sequentially consistent* iff there exists a string $\sigma' \in S$ such that, for all processes p, $\sigma \uparrow p$ equals $\sigma' \uparrow p$. A sequentially consistent implementation is any language I over the events Σ, such that all strings in I are sequentially consistent.

An equivalent definition of sequential consistency uses dependency relations. Define the (symmetric) dependency relation sc over Σ to contain all the pairs $o((p, A, w, v), o'(p', A', w', v'))$ such that $p = p'$. Thus, operations of the same process are dependent, and those of different processes are commutable. By definition, a string σ is sequentially consistent iff $\sigma \in cl_{sc}(S)$. Checking whether an implementation I is sequentially consistent with respect to S reduces to checking

$$I \subseteq cl_{sc}(S).$$

One case where sequential consistency is commonly used is in the specification of shared memory systems. In this case, a finite state protocol is used to maintain the contents of local cache memories in such a way that loads and stores appear sequentially consistent to the programmer. In this case, each memory address is an atomic read/write object, and each processor accessing the shared memory is a concurrent process. For a fixed number of memory addresses, the implementation is finite-state and thus the language of the implementation is regular. We might therefore hope to verify the protocol for the case of a small number of processors and addresses by using a model checking approach. It is known that the general problem $I \subseteq cl_{sc}(S)$ is undecidable [AH89]. In Section 3, we show that the problem remains undecidable even in the special case when the specification S is a collection of atomic read/write objects (however, we leave open the possibility that an algorithm exists for some fixed number of objects less than four). Undecidability implies that the language $cl_{sc}(S)$ of sequentially consistent strings is not regular (in fact, it is not even context-free); thus any finite state implementation that is sequentially consistent obeys some property that is stronger. For verification purposes, it may therefore be more appropriate to use a specification that is stronger than sequential consistency *per se*.

2.3. Linearizability

Linearizability was introduced by Herlihy and Wing [HW90] as a stronger requirement than sequential consistency.

Concurrent Implementations of Objects

The specification of an object assumes that the operations are instantaneous or atomic. In an actual implementation, each operation spans over a period of time and may involve a sequence of steps. For instance, the specification of a *stack* asserts the legal sequences of *push* and *pop* operations. In an actual implementation, a single *push* operation may correspond to a series of steps that invoke operations on simpler objects such as registers and arrays. Furthermore, when processes accessing the object run concurrently, different operations may execute concurrently. Hence, we split each operation into an invocation and a return.

Given an object A, for each process p, an operation o, and an input w, let $o_i(p, A, w)$ denote the event that the process p invokes the code that implements the operation o on object A with input w. For a response v, let $o_r(p, A, v)$ denote the event that for the process p, the execution of the operation o on object A returns with response v. The set of all invocation events of the object A is denoted by $\Sigma^i(A)$, its set of response events by $\Sigma^r(A)$, and their union by $\Sigma^{ir}(A)$. The union of such events over all objects is denoted Σ^{ir}. The set of invocation and response events belonging to a single process p is denoted by $\Sigma^{ir}(p)$. Let $\Sigma^{ir}(p, A) = \Sigma^{ir}(p) \cap \Sigma^{ir}(A)$.

A concurrent implementation is a language over the alphabet Σ^{ir}. While operations by different processes may execute concurrently, an individual process accesses the object in a sequential fashion; that is, the invocation and response events of a single process alternate. For each process p, the language

$$[\, +_{A,o} (+_w\, o_i(p, A, w) \cdot +_v\, o_r(p, A, v))]^*$$

over the alphabet $\Sigma^{ir}(p)$ is denoted by $L^{ir}(p)$. A string σ over the alphabet Σ^{ir} is *well formed* iff for every process p, $\sigma \uparrow \Sigma^{ir}(p)$ is in the language $L^{ir}(p)$. A *concurrent implementation* is a language I consisting of well-formed strings over the alphabet Σ^{ir}.

Linearizability Definition

Recall that S is the asynchronous product of specifications of individual objects. Let S^{ir} be the language over the alphabet Σ^{ir} obtained by replacing every symbol $o(p, A, w, v)$ by the string $o_i(p, A, w)\, o_r(p, A, v)$. Every string in S^{ir} is well formed with responses immediately following the invocations. Such strings are called *sequential*. The (asymmetric) dependency relation lin is defined to be

$$\bigcup_p [\Sigma^{ir}(p) \times \Sigma^{ir}(p)] \cup [\Sigma^r \times \Sigma^i].$$

Thus, for two events $a, b \in \Sigma^{ir}$, (a, b) is in the dependency relation *lin* either iff both events belong to the same process or if the first event is a response and the second one is an invocation. A well-formed string σ over the alphabet Σ^{ir} is *linearizable* with respect to the specification S iff σ belongs to the closure $cl_{lin}(S^{ir})$.

Intuitively, a string is linearizable if the invocations and responses can be commuted to obtain a sequential string in the specification. The dependency relation ensures that if two operations belonging to different processes overlap then they may appear in either order in the sequential string, but if the response of one precedes the invocation of the other, then no commuting is possible.

EXAMPLE. For the atomic bit x, Fig. 2 shows both a linearizable and a nonlinearizable string. The left end-point of each interval marks the invocation and the right end-point marks the response. Thus, the string σ is

$$read_i(p_1), write_i(p_2, 1), read_i(p_3), read_r(p_1, 1), write_r(p_2), read_r(p_3, 0).$$

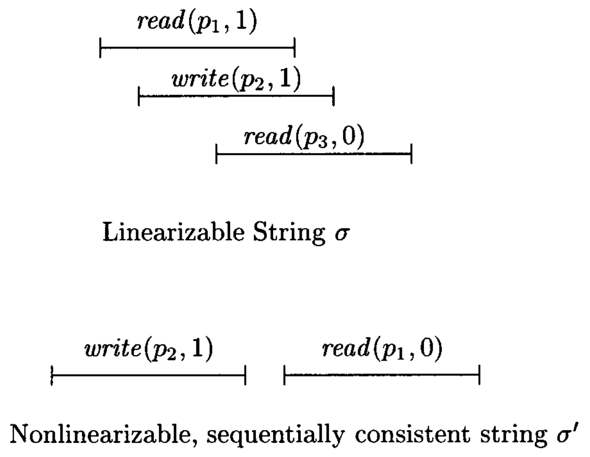
$read(p_1, 1)$
$write(p_2, 1)$
$read(p_3, 0)$
Linearizable String σ
$write(p_2, 1)$
$read(p_1, 0)$
Nonlinearizable, sequentially consistent string σ'

FIG. 2. Sample strings for atomic bit.

It is linearizable because we can commute its operations to obtain the sequential string

$$read_i(p_3), read_r(p_3, 0), write_i(p_2, 1), write_r(p_2), read_i(p_1), read_r(p_1, 1)$$

which meets the specification. On the other hand, in the string σ'

$$write_i(p_2, 1), write_r(p_2), read_i(p_1), read_r(p_1, 0)$$

no commuting is possible, since the two operations do not overlap, and the string is not linearizable.

The concurrent implementation I is *linearizable* iff every string in I is linearizable. Thus, checking linearizability of an implementation I corresponds to checking language-inclusion $I \subseteq cl_{\text{lin}}(S^{ir})$.

It turns out that linearizability, unlike sequential consistency, can be checked separately for individual objects [HW90]:

LEMMA 2. *A well-formed string σ over Σ^{ir} is linearizable iff for every object A, $\sigma \upharpoonright \Sigma^{ir}(A)$ is linearizable.*

Formulation Using Commit Points

An alternative formulation of linearizability uses the notion of *commit* points. A well-formed string σ is linearizable if we can insert between every pair of matching invocation $o_i(p, A, w)$ and response $o_r(p, A, v)$ the operation $o(p, A, w, v)$ such that the projection of the resulting string on the events in Σ is in the specification language S. To formalize this intuition, consider the joint alphabet $\Sigma \cup \Sigma^{ir}$, denoted by Σ^{oir}. As before, the subset of Σ^{oir} containing events by a single process p is denoted by $\Sigma^{oir}(p)$. For each process p, the language

$$[\, +_{A,o} +_w (o_i(p, A, w) \cdot +_v (o(p, A, w, v) \cdot o_r(p, A, v)))]^*$$

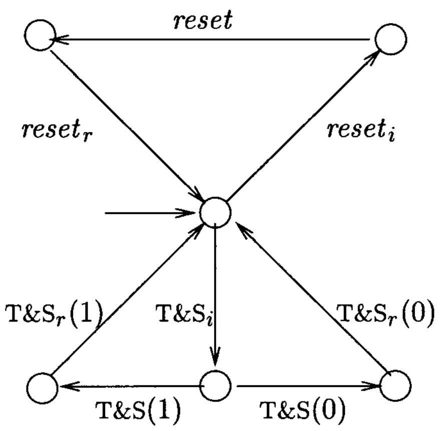

FIG. 3. The language of invocations, commits, and responses for test-and-set.

over the alphabet $\Sigma^{oir}(p)$ is denoted by $L^{oir}(p)$. The language $L^{oir}(p)$ corresponding to a test-and-set bit is shown in Fig. 3.

LEMMA 3. *A well-formed string σ over the alphabet Σ^{ir} is linearizable with respect to the specification S iff there exists a string τ over the alphabet Σ^{oir} such that the following three constraints are satisfied*:

(C1) *The projection $\tau \upharpoonright \Sigma^{ir}$ equals σ.*

(C2) *For each process p, the projection $\tau \upharpoonright \Sigma^{oir}(p)$ is in the language $L^{oir}(p)$.*

(C3) *The projection $\tau \upharpoonright \Sigma$ belongs to the specification S.* ∎

Proof. Given a string σ over Σ^{ir}, let us call a string τ over Σ^{oir} a *witness* for σ if τ satisfies the conditions C1, C2, and C3. We wish to establish that a string is linearizable iff it has a witness.

Only if direction. Assume that σ is linearizable. Then $\sigma = \sigma_n \Rightarrow_{\text{lin}} \sigma_{n-1} \Rightarrow_{\text{lin}} \cdots \Rightarrow_{\text{lin}} \sigma_0$ for some sequential string σ_0 in S^{ir}. We prove that all the strings σ_i in this sequence have a witness by induction.

Consider the sequential string $\sigma_0 \in S^{ir}$. Insert the commit event $o(p, A, w, v)$ between every adjacent pair $o_i(p, A, w)$ and $o_r(p, A, v)$. The resulting string τ satisfies C1, C2, and C3 and hence is a witness for σ_0.

Now consider $\sigma_{i+1} \Rightarrow_{\text{lin}} \sigma_i$. Let $\sigma_{i+1} = \rho_1 \cdot ab \cdot \rho_2$ and $\sigma_i = \rho_1 \cdot ba \cdot \rho_2$ obtained from σ_{i+1} by commuting a and b. By induction hypothesis, σ_i has a witness, and there exists a string τ over Σ^{oir} satisfying C1, C2, and C3. By C1, $\tau = \tau_1 \cdot b \cdot \tau_2 \cdot a \cdot \tau_3$ such that τ_2 contains only the commit operations. First observe that if a is an invoke operation then its matching commit lies in τ_3, and we can shift a to the left without destroying C2. Similarly, if b is a response operation, then its matching commit lies in τ_1, and we can shift b to the right without destroying C2. We will construct a witness τ' for σ_{i+1} by modifying τ.

If a and b are invoke operations, then choose $\tau' = \tau_1 \cdot ab \cdot \tau_2 \cdot \tau_3$. If both a and b are response operations, then use $\tau' = \tau_1 \cdot \tau_2 \cdot ab \cdot \tau_3$. If a is invoke and b is response, then choose $\tau' = \tau_1 \cdot a \cdot \tau_2 \cdot b \cdot \tau_3$. In each case, $\tau' \uparrow \Sigma$ equals $\tau \uparrow \Sigma$ and hence belongs to S. $\tau' \uparrow \Sigma^{ir} = (\tau_1 \uparrow \Sigma^{ir}) \cdot ab \cdot (\tau_3 \uparrow \Sigma^{ir})$ and hence $\tau' \uparrow \Sigma^{ir} = \rho_1 \cdot ab \cdot \rho_2 = \sigma_{i+1}$. Furthermore, since a and b belong to different processes, and for each commit operation, the enclosing invoke and response does not change, $\tau' \uparrow \Sigma^{oir}(p) = \tau \uparrow \Sigma^{oir}(p)$, and hence τ' satisfies C2. This establishes that τ' is a witness for σ_i.

If direction. Assume that σ has a witness, and let τ be the corresponding string satisfying C1, C2, and C3. By C1, the string σ embeds within τ. By C2, each invoke and response operation in τ is identified with a unique commit operation. Consequently, each symbol in σ can be identified with a unique commit operation in τ. For $i < j$, the pair (i, j) of positions in σ is said to be out-of-order if, in τ, the commit operation corresponding to the ith symbol appears after the commit operation for the jth symbol. We use induction on the number of out-of-order pairs in σ.

Suppose σ has no out-of-order pairs. Then σ has to be sequential. There is only one possible way of introducing commits in a sequential string, and thus, τ is completely determined by σ. In this case, since τ satisfies C3, $\sigma \in S^{ir}$, and thus, linearizable.

Suppose σ has $k > 0$ out-of-order pairs. Observe that if σ has some out-of-order pair, then some pair of adjacent positions has to be out-of-order. Then, $\sigma = \rho_1 \cdot ab \cdot \rho_2$ such that the commit corresponding to a is after the commit corresponding to b in τ. This can happen only when a and b belong to different processes and are either both invokes, or both responses, or a is invoke and b is response. Thus, $(a, b) \in lin$. Let $\sigma' = \rho_1 \cdot ba \cdot \rho_2$, and we have $\sigma \Rightarrow_{lin} \sigma'$. Since τ is a witness for σ, $\tau = \tau_1 \cdot a \cdot \tau_2 \cdot b \cdot \tau_3$ such that τ_2 contains only the commit operations. We construct a string τ' that is a witness for σ'.

If a and b are invoke operations, then choose $\tau' = \tau_1 \cdot ba \cdot \tau_2 \cdot \tau_3$. If both a and b are response operations, then use $\tau' = \tau_1 \cdot \tau_2 \cdot ab \cdot \tau_3$. Suppose a is invoke and b is response. Let $\tau_2 = \tau_2' \cdot \tau_2''$ such that τ_2' does not contain the commit corresponding to a and τ_2'' does not contain the commit corresponding to b. This is possible since we know that the commit for a is after the commit for b. Choose $\tau' = \tau_1 \cdot \tau_2' \cdot ba \cdot \tau_2'' \cdot \tau_3$. In each case, $\tau' \uparrow \Sigma$ equals $\tau \uparrow \Sigma$ and hence belongs to S. Furthermore, $\tau' \uparrow \Sigma^{ir} = \sigma'$, and for each p, $\tau' \uparrow \Sigma^{oir}(p) = \tau \uparrow \Sigma^{oir}(p)$. This establishes that τ' is a witness for σ'.

The number of out-of-order pairs in σ' with respect to the witness τ' is $k-1$. Hence by induction, σ' is linearizable. Since $\sigma \Rightarrow_{lin} \sigma'$, σ is linearizable. ▮

Recall that sequential consistency corresponds to considering two events to be dependent only when they belong to the same process, equivalently, to replacing C1 by a weaker requirement C1′ which says that for every process p, $\tau \uparrow \Sigma^{ir}(p)$ equals $\sigma \uparrow \Sigma^{ir}(p)$ (i.e., every process sees the same sequence of invocations and responses). Thus, the string σ' of Fig. 2 is sequentially consistent.

The original formulation of linearizability [HW90] allows some pending invocations without a matching response. A string with pending invocations is linearizable if it has a linearizable completion obtained by adding appropriate responses. We consider only strings in which all invocations have been matched, and this leads to simpler definitions. While applying our definition to a distributed implementation, one needs to check, in addition to linearizability, the existential property that every invocation has a possible response, expressed by the CTL-formula:

$$\forall\Box\ (\mathit{involke} \to \exists\Diamond \mathit{response}).$$

2.4. Serializability

Serializability as a correctness criterion for database transactions was first discussed in [EGLT76]. Database transactions are a generalization of operations on atomic objects; the execution of each transaction consists of several operations such as reads and writes to memory objects. An execution of a transaction system is sequential if the occurrences of the transactions are not interleaved, that is, transactions execute in full, one after the other. Database serializability is a correctness criterion for ensuring that database transactions appear to execute in a sequential fashion. The criterion is defined using an equivalence relation among executions. Strings that are equivalent are considered indistinguishable. A system is serializable if every execution is indistinguishable from a sequential execution. For us, the equivalence is defined by a symmetric dependency (conflict) relation among operations (this corresponds to the so-called *conflict-serializability* which is the most broadly used definition among the various definitions appearing in the literature). The transactions can occur multiple times in a single execution, and we allow internal choices in the transactions, which allow them to execute different operations in different incarnations.

A *database system DB* consists of

• A finite set $\mathcal{T}$ of *transactions*. Every transaction T has a finite alphabet Σ_T of operations. We assume that for $T \neq T'$, Σ_T and $\Sigma_{T'}$ are disjoint. Each alphabet Σ_T includes two special symbols $\mathit{begin}(T)$ to mark the beginning of an instance of the transaction and $\mathit{end}(T)$ to mark the end of of the transaction. Denote $\Sigma'_T = \Sigma_T \setminus \{\mathit{begin}(T), \mathit{end}(T)\}$. The set $\Sigma = \bigcup_T \Sigma_T$ contains all events. The *specification* of a transaction T is the regular language $S(T)$, which is required to be a subset of $\mathit{begin}(T)(\Sigma'_T)^*\ \mathit{end}(T)$.

• A finite set of *objects*. Every object A has a finite alphabet Σ_A. It holds that $\bigcup_A \Sigma_A = \bigcup_T \Sigma'_T$, i.e., every event besides begin and end transaction involves some object. The *specification* of an object A is the regular prefix closed language $S(A)$.

• A symmetric dependency relation se satisfying that (1) if $(o(T, A, v, w), o'(T', A', v', w')) \in se$ then either $T = T'$ or $A = A'$ (that is, events can be dependent only if they involve the same transaction or the same object), and (2) every operation of transaction T must be dependent on $\mathit{begin}(T)$ and $\mathit{end}(T)$. We assume that the specifications of the transactions and the objects are closed under the dependency relation, that is, $S(T) = cl_{se}(S(T))$ and $S(A) = cl_{se}(S(A))$.

The definition of a database system allows independency, i.e., concurrency, among events that operate over the same object. This allows, e.g., concurrent reads of the same object. Independence among events of the same transaction is allowed, but is not typical.

An *occurrence* of a transaction is a string from $S(T)$. It begins with the symbol $begin(T)$, followed by a string over Σ'_T, followed by $end(T)$. In the database system, all the transactions run in parallel and occur repeatedly. The *executions* of a database system DB is the asynchronous product $I = (\|_T S(T)^*) \| (\|_A S(A))$.

Observe that the possible interleavings of parallel transactions is constrained by the synchronization introduced by the objects. Intuitively, serializability of a language means that each execution in the language is trace equivalent to one in which occurrences of transactions are executed completely one after the other. Let $SP = +_T (begin(T)\ (\Sigma'_T)^*\ end(T))$. The database DB is *serializable* iff $I \subseteq cl_{se}(SP^*)$.

EXAMPLE. Consider a typical database system, with the following operations:

$rlock(T, x)$ -T locks object x for read only.
$wlock(T, x)$ -T locks object x for write only.
$unlock(T, x)$ -T unlocks object x.

In this case, two operations are *dependent* iff either (1) they belong to the same transaction, or (2) they lock the same object and at least one of them is a write-lock. A typical specification $S(x)$ of the object x is the set of prefixes of:

$$[(\|_T (rlock(T)\ unlock(T))^*)\ +_T (wlock(T)\ unlock(T))]^*$$

That is, many read locks may be held concurrently, but write locks are exclusive. If the database system has two copies T_1 and T_2 of the transaction whose specification contains the single string

$$begin(T)\ wlock(T, x)\ unlock(T, x)\ wlock(T, y)\ unlock(T, y)\ end(T)$$

then it is not serializable, since

$$begin(T_1)\ begin(T_2)\ wlock(T_1, x)\ unlock(T_1, x)\ wlock(T_2, x)\ unlock(T_2, x)$$
$$wlock(T_2, y)\ unlock(T_2, y)\ wlock(T_1, y)\ unlock(T_1, y)\ end(T_1)\ end(T_2)$$

is an execution of DB, which cannot be shuffled such that one of the occurrences of the transactions executes entirely after the other. On the other hand, a database with two copies of the transaction

$$begin(T)\ wlock(T, x)\ wlock(T, y)\ unlock(T, x)\ unlock(T, y)\ end(T),$$

following the well-known two-phase locking protocol, is serializable.

3. UNDECIDABILITY OF SEQUENTIAL CONSISTENCY

We now consider the model checking problem for sequential consistency, where the implementation I is a regular language and S is a specification of a finite collection of finite-state objects. The basic result is that testing $I \subseteq cl_{sc}(S)$ is undecidable, even for the special case of read/write objects. We argue this in two steps. First, we show that $I \subseteq cl_{sc}(S)$ is undecidable for arbitrary regular languages S. This is a special case of a theorem in [AH89]. The proof in [AH89] shows that this problem is undecidable even if we set $I = \Sigma^*$. However, here we offer an alternative proof, which may provide some additional insight into why the problem is undecidable. Next, we show that this problem reduces to the special case where $S = S^{rw}$, the language of read/write objects.

The proof for the general case is in two steps:

- Effectively reduce the n-counter halting problem (which we will denote n-ZN, for "n counters with test for zero and test for nonzero") to n-counter halting without test for nonzero (which will be denoted n-Z).
- Effectively reduce n-Z to $I \subseteq cl_{sc}(S)$, for suitable I and S.

Counter Machines

The control of a counter machine is a finite automaton M, whose alphabet is made up of increment, decrement and test operations. For the case of an n-counter machine with both test for zero and non-zero, let $\Sigma_{n\text{-ZN}}$ be the union $\bigcup_{i=1}^{n} \{\mathsf{I}_i, \mathsf{D}_i, \mathsf{Z}_i, \mathsf{N}_i\}$. The symbols I_i, D_i, Z_i, N_i stand respectively for increment, decrement, test for zero, and test for non-zero on counter i.

We let $c_{\sigma,j}$ denote the value of counter j after the string σ of operations of the finite control (i.e., $c_{\sigma,j}$ is the difference $|\sigma \uparrow \mathsf{I}_j| - |\sigma \uparrow \mathsf{D}_j|$ between the number of increments and decrements). We say that a string σ of the finite control is *admitted* iff

(1) for all prefixes $\pi \mathsf{Z}_j$ of σ, $c_{\pi,j} = 0$, and

(2) for all prefixes $\pi \mathsf{N}_j$ of σ, $c_{\pi,j} \neq 0$.

That is, a string is admitted if whenever counter i is tested for zero it is zero, and whenever it is tested for nonzero it is nonzero. The decision problem n-ZN is to determine, for a given finite automaton M on alphabet $\Sigma_{n\text{-ZN}}$, whether some $\sigma \in L(M)$ is admitted.

LEMMA 4. *n-ZN is undecidable, for $n \geqslant 2$.*

We now reduce this problem to the case without test for nonzero. Let $\Sigma_{n\text{-Z}}$ be the union $\bigcup_{i=1}^{n} \{\mathsf{I}_i, \mathsf{D}_i, \mathsf{Z}_i\}$. The decision problem n-Z is to determine, for a given finite automaton M on alphabet $\Sigma_{n\text{-Z}}$ whether some $\sigma \in L(M)$ is admitted.

THEOREM 1. *n-Z is undecidable, for $n \geqslant 3$.*

Proof. By reducing n-ZN to $(n+1)$-Z. Replace every occurrence of N_i by the following

$$(\mathsf{D}_i(\mathsf{D}_i\mathsf{I}_{n+1})^* \, \mathsf{Z}_i(\mathsf{I}_i\mathsf{D}_{n+1})^* \, \mathsf{Z}_{n+1}\mathsf{I}_i)$$
$$(\mathsf{I}_i(\mathsf{I}_i\mathsf{D}_{n+1})^* \, \mathsf{Z}_i(\mathsf{D}_i\mathsf{I}_{n+1})^* \, \mathsf{Z}_{n+1}\mathsf{D}_i).$$

Note that after executing the above, the values of the counters remain unchanged (since counters i and $n+1$ are incremented and decremented an equal number of times) and counter i must be nonzero (that is, a positive value is decremented until zero and then restored, and similarly a negative value is incremented to zero and then restored). If counter i is zero at the beginning of this sequence, then the given string cannot be admitted. ▮

This result might be of some independent interest for undecidability proofs in general, since it demonstrates a slightly weaker class of machines that are Turing complete (albeit with one additional counter).

Undecidability of the General Case

We now observe that a string is admitted when the number of I_i's and D_i's between any two Z_i's is equal. When we allow I_i and D_i to commute with each other, but not with Z_i, then the number of increments and decrements is equal exactly when they commute to some string in $(\mathsf{I}_i \mathsf{D}_i)^*$. This allows us to reduce the existence of an admitted string to the problem of containment in the closure of a regular language.

LEMMA 5. *Let S and I be regular languages, over some alphabet Σ, and let $D \subseteq \Sigma^2$. The proposition $I \subseteq cl_D(S)$ is undecidable.*

Proof. By reduction from n-Z. Let I be the language of the finite control, and let

$$S = \bigcup_{j=1}^{n} ((\mathsf{I}_j + \mathsf{D}_j)^* \, \mathsf{Z}_j)^* \, (\mathsf{I}_j \mathsf{D}_j)^* \, (\mathsf{I}_j^+ + \mathsf{D}_j^+) \, \mathsf{Z}_j \Sigma_{n\text{-Z}}^*$$

and let D be such that I_i and D_i do not commute with Z_i, but all other pairs commute. The language S is constructed so that its closure is all of the strings that are *not* admitted. Thus, I contains an admitted string exactly when $I \not\subseteq cl_D(S)$. ▮

THEOREM 2. *The problem of checking sequential consistency, for implementation and specification given by regular languages, is undecidable.*

Proof. By reduction from n-Z. Let A be a concurrent object, with one operation o, having no input, and outputs in the set $\Sigma_{n\text{-Z}}$. We use the same languages I and S as in the previous proof,[4] except that we make the following substitutions:

$$\mathsf{I}_i \rightarrow o(p_{2i}, A, \mathsf{I}_i)$$

$$\mathsf{D}_i \rightarrow o(p_{2i+1}, A, \mathsf{D}_i)$$

$$\mathsf{Z}_i \rightarrow o(p_{2i}, A, \mathsf{Z}_i) \; o(p_{2i+1}, A, \mathsf{Z}_i).$$

[4] *Note.* This argument is somewhat oversimplified, since the language S is not prefix closed. However, both I and S can be made prefix closed by appending a special "termination" operation to every string and allowing all strings without terminators in S.

In this way, we obtain the desired dependence relation between the encodings of I_i, D_i, and Z_i. Now the finite control contains an admitted string exactly when $I \not\subseteq cl_D(S)$. ▮

Special Case of Read/Write Registers

We now consider the problem when the operations on the concurrent objects are restricted to reads and writes. That is, for any object A, let Σ_A^{rw} be the set of all operations $read(p, A, v)$ and $write(p, A, v)$ for some process p and value v. Let the specification S_A^{rw} be the set of strings $\sigma \in \Sigma_A^{rw}$ where each value read matches the most recent write; that is, if $\pi \cdot read(p, A, v)$ is a prefix of σ, then

$$\pi \in (\Sigma_A^{rw})^* \, write(A, v) \, read^*$$

Let $S^{rw} = \|_A \, S_A^{rw}$.

THEOREM 3. *The problem of checking $I \subseteq cl_{sc}(S^{rw})$, where I is a regular language, is undecidable.*

Proof. This theorem can be proved by a reduction from the previous problem, namely, deciding whether $I \subseteq cl_D(S)$. Since this reduction is fairly complex, it will help to introduce some notation in order to describe it more succinctly. When a sequence of operations is in S^{rw}, we will say that it is totally consistent. We will first of all need to be able to define sequences of operations that are atomic, in the sense that atomic sequences can never be interleaved in a totally consistent string. To accomplish an atomic sequence of operations, a process first writes its own identifier into a designated memory location A_b, then performs the sequence of operations, then reads its own identifier from A_b. If any other process has begun an atomic sequence in the interim, then the final read of A_b will be inconsistent. We will use the following notation for such a "bracketed" sequence of operations,

$$\{\pi\}_p = w(p, A_b, p) \, \pi \, r(p, A_b, p),$$

where π is a sequence of operations of process p. In effect, a bracketed sequence of this form acts like an atomic operation of process p and commutes with atomic operations of other processes. As an additional convenience, for any object drawn from some finite set (e.g., a state of an automaton) we will use square brackets to denote a unique integer encoding that object. The symbol ϵ will stand for some otherwise unused integer.

Now, the basic idea of the proof is as follows: we start with a problem $I \subseteq cl_{sc}(S)$, where I and S are languages defined by finite automata. We can assume without loss of generality that the automaton for S has a unique final state (if not, we can append to all strings in I and S a special terminal symbol that takes the corresponding automata to a unique final state and commutes with all other symbols). Given the automaton for S, we will define a function ϕ that maps every string σ to a string σ' such that $\sigma \in cl_{sc}(S)$ exactly when $\sigma' \in cl_{sc}(S^{rw})$.

The intuition behind ϕ is as follows. Let $\mathscr{S}$ be the automaton accepting S. The map ϕ takes each symbol α in a string σ and maps it to a string of operations encoding α itself and all possible transitions of $\mathscr{S}$ on α. The resulting string σ' can be shuffled to a totally consistent string only by constructing, as a prefix, an accepting run of $\mathscr{S}$ on some shuffle of σ and shuffling all the unused transitions to the end of the string. Thus σ' is sequentially consistent w.r.t. S^{rw} exactly when σ is sequentially consistent w.r.t. S.

Now we define the precise encoding of alphabet symbols and transitions in $\phi(\sigma)$. We represent a transition $\langle s, \alpha, s' \rangle$ of $\mathscr{S}$ by a sequence of reads and writes as follows:

$$\rho(\langle s, \alpha, s' \rangle) = \{r(A_s, [s])\, r(A_l, [\alpha])\, w(A_s, [s'])\, w(A_l, [\epsilon])\}_{[s, \alpha, 0]}.$$

The two read operations act as a guard, guaranteeing that the current state is s (stored in location A_s) and the current tape symbol is α (stored in location A_l) before the transition starts. After the transition, the current state is s' and the tape symbol is "used up" by setting it to ϵ.

Each alphabet symbol α, an operation of process p, is represented by a sequence

$$\gamma(\alpha) = \{r(A_l, \epsilon)\, r(A_e, 0)\, w(A_l, [\alpha])\}_{[p]}.$$

The first read operation ensures that the previous tape symbol has been used up, the second read ensures that symbols cannot commute past the "end marker," and the write sets the current symbol (in location A_l) to α. Note that transitions and tape symbols commute because they are operations of different processes, but they cannot interleave because they are bracketed. Note also that tape symbols and transitions must alternate in any totally consistent string. For each tape symbol α in σ, we will insert into σ' a copy of α and of every transition $\langle s, \alpha, s' \rangle$ in $\mathscr{S}$. In a totally consistent shuffle of σ', we want one transition to follow each tape symbol and the remainder to shuffle to the end of the string. To allow this, we also insert a collection of placeholders, one per transition, that must also shuffle to the end of the string. The placeholder for a transition $\langle s, \alpha, s' \rangle$ is

$$\rho'(\langle s, \alpha, s' \rangle) = \{r(A_e, 1)\, w(A_s, [s])\, w(A_l, [\alpha])\}_{[s, \alpha, 1]}.$$

The read operation ensures that placeholders must commute past the end marker. Thus, each α in σ translates to

$$\psi(\alpha) = \gamma(\alpha)\, \rho(t_1)\, \rho'(t_1) \cdots \rho(t_k)\, \rho'(t_k),$$

where $\{t_1, ..., t_k\}$ is the set of transitions of $\mathscr{S}$ on α. Finally, we need the "begin marker" π_0, which sets up the initial state of $\mathscr{S}$, and the end marker π_f, which checks the final state. Let

$$\pi_0 = \{r(A_s, \epsilon)\, r(A_l, \epsilon)\, r(A_e, 0)\, w(A_s, [s_0])\}_v$$
$$\pi_f = \{r(A_s, s_f)\, w(A_e, 1)\}_v,$$

where s_0 and s_f are the initial and final states of S, respectively, and v is some otherwise unused process number. Note that a value 1 in memory location A_e is used to indicate that the end marker has passed. All the "placeholders" for unused transitions must shuffle to a point after the end marker, since they read a value 1 from this location. We can now define our map ϕ on strings as follows:

$$\phi(\sigma) = \pi_0 \psi(\sigma_1)\, \psi(\sigma_2) \cdots \psi(\sigma_n)\, \pi_f.$$

If $\phi(\sigma)$ has a totally consistent shuffle, then clearly $\sigma \in cl_{sc}(S)$, since the transitions used in the shuffle define an accepting run of $\mathcal{S}$. Conversely, if σ shuffles to a string with an accepting run in $\mathcal{S}$, we can construct a totally consistent shuffle of $\phi(\sigma)$ by shifting the transitions not used in the run to the end of the string, each after its corresponding placeholder.

From an automaton for a given regular language I, we can effectively construct an automaton for $I' = \{\phi(\sigma) \mid \sigma \in I\}$. This involves only concatenation of the begin and end markers, and substitution of the string $\psi(\alpha)$ for each symbol α (if this is not obvious, imagine performing the same operation on regular expressions instead). Thus, the undecidable problem $I \subseteq cl_{sc}(S)$ can be effectively reduced to the problem $I' \subseteq cl_{sc}(S^{rw})$, which is therefore also undecidable. ▮

Note that the above reduction uses a total of four read/write objects. This is the simplest reduction that we are aware of, leaving open the possibility that sequential consistency may be decidable for a fixed number of read/write objects up to three.

4. DECIDING LINEARIZABILITY

In this section, we consider the problem of verifying that a concurrent implementation I is linearizable with respect to the specification S, that is, checking the inclusion $I \subseteq cl_{\text{lin}}(S^{ir})$.

Computing the Closure

Lemma 3 suggests an equivalent formulation of the language $cl_{\text{lin}}(S)$. Define $C(S)$ to be the language consisting of strings over Σ^{oir} satisfying the requirements C2 and C3 of Lemma 3. Then, Lemma 3 can be reformulated as

$$cl_{\text{lin}}(S^{ir}) = C(S) \uparrow \Sigma^{ir}.$$

The next lemma shows that the language $C(S)$ can be expressed conveniently as an asynchronous product:

LEMMA 6. *$C(S)$ equals $(\|_p\, L^{oir}(p)) \parallel S$.*

Proof. Follows from the definition of the asynchronous product $\|$. ▮

Thus, for a string to be in $C(S)$, the projection on operations of a single process consists of alternation of invocation, commit, and response, and the possible interleaving of the operations of different processes is constrained by the fact that the specification S allows only certain sequences of commits. It follows that

THEOREM 4. *If Σ^{ir} is finite and S is regular then the set $cl_{\mathrm{lin}}(S^{ir})$ of linearizable strings is a regular language.*

Proof. Suppose Σ^{ir} is finite. Then, for each process p, the language $L^{oir}(p)$ is regular. The asynchronous product preserves regularity, and hence, $C(S)$ is regular. The projection operation also preserves regularity, and hence, $C(S)\uparrow\Sigma^{ir}$ is regular. ∎

Complexity

Linearizability can be checked separately for each object. Suppose A is a finite-state object with size k.

LEMMA 7. *For each process p, the alphabet $\Sigma^{oir}(p)$ contains at most $2k^2+k^3$ symbols. The language $L^{oir}(p)$ is regular and can be generated by a DFA with at most $2k^2+1$ states.*

Proof. Assume that there are at most k possible operations; each operation can have at most k possible arguments and can return at most k possible responses. Define a DFA M as follows. There is a single initial state s. For each event $o(p, w, v)$, (1) there is a state $t_{o,w}$ and a state $u_{o,v}$, and (2) there is an edge from s to $t_{o,w}$ labeled with the invoke operation $o_i(p, w)$, an edge from $t_{o,w}$ to $u_{o,v}$ labeled with $o(p, w, v)$, and an edge from $u_{o,v}$ to the initial state s labeled with the response $o_r(p, v)$. The only accepting state is s. The automaton M accepts the language $L^{oir}(p)$ (see Fig. 3 for the automaton corresponding to the test-and-set bit). The number of invoke operations is at most k^2, the number of commit operations is at most k^3, and the number of response operations is at most k^2. The number of states of M is at most $2k^2+1$. ∎

If a language L_1 is generated by an NFA with m_1 states and L_2 is generated by an NFA with m_2 states, then there is an algorithm to construct an NFA, with at most $m_1 \cdot m_2$ states, that accepts the asynchronous product $L_1 \parallel L_2$. If a language L over Σ is generated by an NFA with m states, and Σ' is a subset of Σ, then there is an algorithm to construct an NFA, also with m states, that accepts the projection $L\uparrow\Sigma'$. Putting these together, we get:

LEMMA 8. *If the object A has size k with n processes, then the size of the alphabet Σ^{oir} is bounded by $2nk^2+nk^3$, and the language $cl_{lin}(S^{ir})$ is generated by an NFA with at most $k \cdot 2^n \cdot k^{2n}$ states.*

To check the language-inclusion $I \subseteq cl_{lin}(S^{ir})$, we construct the NFA accepting $cl_{lin}(S^{ir})$, complement it, and test if the intersection of the complement with I is empty. Complementing an NFA involves an exponential subset-construction. This gives a doubly exponential upper bound for checking linearizability:

THEOREM 5. *Let A be an object of size k with n processes, and let I be a concurrent implementation of A given by an NFA with m states. Then the problem of checking whether I is linearizable can be solved in time $O(m \cdot 2^{k \cdot 2^n \cdot k^{2n}})$.*

The space complexity of the above linearizability test is EXPSPACE. This is because the emptiness of the product of I and the complement of the NFA accepting $cl_{lin}(S^{ir})$ can be checked on-the-fly, without explicitly constructing the complement. It is easy to show that the problem is PSPACE-hard; but we do not have a matching EXPSPACE lower bound.

5. DECIDING SERIALIZABILITY

We consider now the algorithm and the complexity of checking serializability. As in the definition, $\mathcal{T}$ is the set of transactions, Σ is the set of events, and se is a symmetric dependency relation, and let $SP = +_T (begin(T)(\Sigma'_T)^* \; end(T))$.

In order to check whether $I \subseteq cl_{se}(SP^*)$ we can generate an automaton M for the complement of $cl_{se}(SP^*)$ and check whether $I \cap L(M)$ is empty. The algorithm for checking whether a fixed string is serializable constructs a graph over the transaction instances and checks for conflict-cycles. This suggests the following construction for M. The automaton M remembers in its finite control the dependency order between active transactions (a transaction is active if it has started but has not yet ended). It also remembers which operations have occurred in the active transactions. When an operation α occurs in some active transaction, it is checked against the operations that occurred in other active transactions. Then an ordering edge is added from any transaction in which an operation β has occurred such that $(\beta, \alpha) \in se$ to the transaction which includes α. Then edges are added to form a transitively closed relation. A cycle in this order means that the string is not serializable.

Assume that there are n transactions $T_1 \ldots T_n$. The serializability automaton M has the following components:

- State-space is $2^{(1..n) \times (1..n)} \times 2^{\Sigma_{T_1}} \times \cdots \times 2^{\Sigma_{T_n}}$. The first component of each state s, denoted $PO(s)$, consists of a transitive relation on elements labeled $1 \ldots n$ (it denotes the conflict dependencies among the active transactions). If the string σ being read is serializable, then there must be a trace equivalent string $\tau \equiv_{se} \sigma$ in which the occurrences of transactions that have started but not yet ended are ordered according to $PO(s)$. For each transaction T_i, there is a component, denoted $\Sigma_i(s)$, that is a subset of Σ_{T_i}. These are the operations which occurred in the current active execution of the transaction T_i.

- The initial state is s_0 with $PO(s_0)$ the empty relation and $\Sigma_i(s_0)$ the empty set for $1 \leqslant i \leqslant n$.

- The transition function δ maps each state s and operation α to the next state s' according to the following cases: If $PO(s)$ contains a cycle then $s' = s$. Otherwise, let α be an operation of transaction T_i, and consider the following two cases:

— α does not end a transaction: Set $\Sigma_i(s')$ to $\Sigma_i(s) \cup \{\alpha\}$, and $PO(s')$ to the *transitive closure* of the relation

$$PO(s) \cup \{(j, i) \mid \exists \beta(\beta \in \Sigma_j(s) \wedge (\beta, \alpha) \in se)\}.$$

That is, add α to the operations of transaction T_i, and add all edges induced by the occurrence of α. Note that the application of the transitive closure is crucial to the construction.

— α is the end-transaction event end_{T_i}: Set $PO(s')$ to $PO(s) \setminus \{(j, k) \mid j = i \vee k = i\}$, and $\Sigma_i(s')$ to empty set. That is, since there is no active occurrence of T_i any more, we remove all the edges between i and other nodes and remove the operations that appeared in T_i.

- The accepting states are those states s in which $PO(s)$ is acyclic, i.e., a partial order, and $\Sigma_i(s)$ is empty set, for $1 \leqslant i \leqslant n$.

THEOREM 6. *The automaton M accepts exactly the language $cl_{se}(SP^*)$.* ∎

This gives:

THEOREM 7. *Suppose I is given by an NFA with m states, there are n transactions, and for each transaction T, Σ_T contains at most k operations. Then, the problem of checking whether I is serializable can be solved in time $O(m \cdot 2^{n^2+nk})$.* ∎

The space complexity of the above test for serializability is PSPACE. Checking serializability is PSPACE-complete in the number of processes, i.e., the totality of transactions and objects. The hardness follows from the hardness of reachability. Notice that the construction of the automaton M works also in the case that se is not symmetric.

6. CONCLUSION

We have considered three problems in the verification of systems of concurrent objects that can be stated in the form $I \subseteq cl_D(S)$, for appropriate regular languages I and S and appropriate dependency relations D. This formalization provides some perspective on the similarities and differences between the three notions of correctness. All are based on the idea of serializing a string, which means shuffling its operations so that transactions appear to be sequential. Sequential consistency and linearizability require that a string be serialized so that it meets a given specification. The difference is in which operations are allowed to commute. For serializability, on the other hand, we require only that the string can be serialized at all. In this case the dependency relation is instance specific.

In two cases, linearizability and serializability, the closure $cl_D(S)$ was found to be regular. The reasons for this are different in each case. In the case of linearizability, the inability to commute nonoverlapping operations means that any given transaction can commute over only a finite number of other transactions (those it overlaps with). Thus, the closure under commutation can be recognized with finite memory. In the case of serializability, it is only necessary to commute each transaction over a finite number of other transactions (again, those it overlaps with) in order to reach the "nearest" string in S. Commutation of nonoverlapping transactions is not needed because the language S permits the transactions to occur in any order. This also explains why, in the case of serializability, the closure is of size that is only singly exponential, instead of doubly exponential. The language S is such that there is no need to guess the order of commit points—all orders are allowable. The automaton obtained for the closure is therefore deterministic and easily complementable.

There are some implications of these results for automatic verification of systems of concurrent objects. First, it is clearly preferable to check linearizability rather than the weaker condition of serializability when both are applicable. When checking linearizability, the exponential space complexity might be avoided by requiring the user to provide a deterministic automaton that fixes the commit point for each operation. There are some applications (such as cache coherence) where linearizability is not applicable because commit points for operations cannot be found in the range of time over which those operations are pending. In this case, we know that any finite state implementation must satisfy some property that is stronger than sequential consistency, since the language of sequentially consistent strings is nonregular. In this case, one might consider specifying some stronger, but regular property to allow for more efficient verification.

Left open by this work are the questions of lower bounds for model checking serializability and linearizability.

ACKNOWLEDGMENTS

We thank Patrice Godefroid, Tom Henzinger, Michael Merritt, and Mihalis Yannakakis for helpful discussions.

REFERENCES

[AH89] Aalbersberg, I. J., and Hoogeboom, H. J. (1989), Characterizations of the decidability of some problems for regular trace languages, *Math. Systems Theory* **22**, 1–19.

[BHG87] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987), "Concurrency control and Recovery in Database Systems," Addison–Wesley, Reading, MA.

[EGLT76] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. (1976), The notions of consistency and predicate locks in a relational database system, *Commun. Assoc. Comput. Mach.* **8**, 624–633.

[FR85] Flé, M. P., and Roucairol, G. (1985), Maximal serializability of iterated transaction, *Theoret. Comput. Sci.* **38**, 1–16.

[GK92] Gibbons, P., and Korach, E. (1992), The complexity of sequential complexity, *in* "Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing," pp. 317–325.

[HW90] Herlihy, M. P., and Wing, J. M. (1990), Linearizability: A correctness condition for concurrent objects, *ACM Trans. Progrg. Lang. Systems* **12**, 463–492.

[Lam79] Lamport, L. (1979), How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Trans. Comput.* **28**, 690–691.

[LMWF94] Lynch, N., Merritt, M., Weihl, W., and Fekete, A. (1994), "Atomic Transactions," Morgan Kaufmann, San Mateo, CA.

[Maz87] Mazurkiewicz, A. (1987), Trace theory, *in* "Advances in Petri Nets: Proceedings of an Advanced Course," Lecture Notes in computer Science, Vol. 255, pp. 279–324, Springer-Verlag, Berlin.

[Och95] Ochmaǹski, E. (1995), Recognizable trace languages, *in* "The Book of Traces" (V. Diekert and G. Rozenberg, Eds.), pp. 67–204, World Scientific, Singapore.

[Pap86] Papadimitriou, C. H. (1986), "The Theory of Database Concurrency Control," Computer Science Press, New York.