

the language of sequentially consistent strings is non-regular. 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.

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

## References

- [BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a relational database system. *Communications of the ACM*, 8:624–633, 1976.
- [FR85] M.P. Flé and G. Roucairol. Maximal serializability of iterated transaction. *Theoretical Computer Science*, 38:1–16, 1985.
- [GK92] P. Gibbons and E. Korach. The complexity of sequential complexity. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 317–325, 1992.
- [HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
- [Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
- [LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.
- [Maz87] A. Mazurkiewicz. Trace theory. In *Advances in Petri nets: Proceedings of an advanced course*, LNCS 255, pages 279–324. Springer-Verlag, 1987.
- [Och95] E. Ochmański. Recognizable trace languages. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 67–204. World Scientific Press, 1995.
- [Pap86] C.H. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.

trace equivalent word  $\tau \equiv_{se} \sigma$  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  $\Sigma_{T_i}$ . 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  $1 \leq i \leq n$ .
- The transition function  $\delta$  maps each state  $s$  and operation  $\alpha$  to the next state  $s'$  according to the following cases: If  $PO(s)$  contains a cycle then  $s' = s$ . Else let  $\alpha$  be an operation of transaction  $T_i$ , and consider the following two cases

- $\alpha$  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  $\alpha$  to the operations of transaction  $T_i$ , and add all edges induced by the occurrence of  $\alpha$ . Note that the application of the transitive closure is crucial to the construction.

- $\alpha$  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  $1 \leq i \leq n$ .

**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 transaction  $T$ ,  $\Sigma_T$  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 non-overlapping 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” sequence in  $S$ . Commutation of non-overlapping 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

## 4 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  $I \subseteq cl_{lin}(S^{ir})$ .

### Computing the closure

Lemma 2 suggests an equivalent formulation of the language  $cl_{lin}(S)$ . Define  $C(S)$  to be the language consisting of strings over  $\Sigma^{oir}$  satisfying the requirements  $C2$  and  $C3$  of Lemma 2. Then,

**Lemma 5**  $cl_{lin}(S)$  equals  $C(S) \uparrow \Sigma^{ir}$ . ■

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

**Lemma 6**  $C(S)$  equals  $(\parallel_p L^{oir}(p) \parallel S)$ . ■

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  $\Sigma^{ir}$  is finite and  $S$  is regular then the set  $cl_{lin}(S^{ir})$  of linearizable strings is a regular language. ■

### 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. ■

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  $\Sigma$  is generated by an NFA with  $m$  states, and  $\Sigma'$  is a subset of  $\Sigma$ , 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  $\Sigma^{oir}$  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 Serializability

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

In order to check whether  $I \subseteq cl_{se}(SP^*)$  we can generate an automaton  $M$  for the complement language  $\overline{cl_{se}(SP^*)}$  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  $\alpha$  occurs in some active transaction, it is checked against the operations occurred in other active transactions. Then an ordering edge is added from any transaction in which an operation  $\beta$  has occurred such that  $(\beta, \alpha) \in se$  to the transaction which includes  $\alpha$ . 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  $T_1 \dots T_n$ . The serializability automaton  $M$  has the following components:

- State-space is  $2^{(1..n) \times (1..n)} \times 2^{\Sigma_{T_1}} \times \dots \times 2^{\Sigma_{T_n}}$ . The first component of each state  $s$ , denoted  $PO(s)$ , consists of a transitive relation on elements labeled  $1..n$  (it denotes the conflict dependencies among the active transactions). If the string being read  $\sigma$  is serializable, then there must be a

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

**Lemma 3**  $n$ -ZN is undecidable, for  $n \geq 2$ . ■

We now reduce this problem to the case without test for non-zero. Let  $\Sigma_{n\text{-Z}}$  be the union  $\cup_{i=1}^n \{I_i, D_i, Z_i\}$ . The decision problem  $n$ -Z is to determine, for a given finite automaton  $M$  on alphabet  $\Sigma_{n\text{-Z}}$  whether some  $\sigma \in \mathcal{L}(M)$  is halting.

**Theorem 1**  $n$ -Z is undecidable, for  $n \geq 3$ .

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

$$\begin{aligned} & (D_i(D_i I_{n+1})^* Z_i (I_i D_{n+1})^* Z_{n+1} I_i) \\ & \quad + \\ & (I_i (I_i D_{n+1})^* Z_i (D_i I_{n+1})^* Z_{n+1} D_i) \end{aligned}$$

Notice 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 non-zero (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 word cannot be halting. ■

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 sequential consistency

We now observe that a string is halting 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  $(I_i D_i)^*$ . This allows us to reduce the existence of a halting string to the problem of containment in the closure of a regular language.

**Lemma 4** Let  $S$  and  $I$  be regular languages, over some alphabet  $\Sigma$ , and let  $D \subseteq \Sigma^2$ . The proposition  $I \subseteq cl_D(S)$  is undecidable.

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

$$S = \bigcup_{j=1}^n ((I_j + D_j)^* Z_j)^* (I_j D_j)^* (I_j^\dagger + D_j^\dagger) Z_j \Sigma_{n\text{-Z}}^*$$

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 words that are *not* halting. Thus,  $I$  contains a halting word 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 sketch. 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<sup>2</sup>, except that we make the following substitutions:

$$\begin{aligned} I_i & \rightarrow o(p_{2i}, A, I_i) \\ D_i & \rightarrow o(p_{2i+1}, A, D_i) \\ Z_i & \rightarrow o(p_{2i}, A, Z_i) \quad o(p_{2i+1}, A, Z_i) \end{aligned}$$

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 a halting word exactly when  $I \not\subseteq cl_D(S)$ . ■

### 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  $\Sigma_A^{rw}$  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  $\sigma$ , then

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

Let  $S^{rw} = \parallel_A S_A^{rw}$ .

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

This theorem can be proved by a reduction from the previous problem. The construction is too involved to be presented here, however, and will appear in a later version of the paper. The simplest reduction that we are aware of makes use of four atomic registers in the alphabet of  $I$ . This leaves open the possibility that sequential consistency may be decidable for a fixed number of registers up to three.

<sup>2</sup>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  $S$ .

- 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  $begin(T)$  and  $end(T)$ . 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 independence, 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 word from  $S(T)$ . It begins with the letter  $begin(T)$ , followed by a string over  $\Sigma'_T$ , 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  $I = (\parallel_T S(T)^*) \parallel (\parallel_A S(A))$ . 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  $I \subseteq cl_{se}(SP^*)$ .

For 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:

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

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

$$\begin{aligned} &begin(T) \ wlock(T, x) \ unlock(T, x) \\ &wlock(T, y) \ unlock(T, y) \ end(T) \end{aligned}$$

then it is not serializable, since

$$\begin{aligned} &begin(T_1) \ begin(T_2) \ wlock(T_1, x) \\ &unlock(T_1, x) \ wlock(T_2, x) \ unlock(T_2, x) \\ &wlock(T_2, y) \ unlock(T_2, y) \ wlock(T_1, y) \\ &unlock(T_1, y) \ end(T_1) \ end(T_2) \end{aligned}$$

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

$$\begin{aligned} &begin(T) \ wlock(T, x) \ wlock(T, y) \\ &unlock(T, x) \ unlock(T, y) \ end(T) \end{aligned}$$

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

### 3 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. The proof 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 non-zero”) to  $n$ -counter halting without test for non-zero (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  $\cup_{i=1}^n \{I_i, D_i, Z_i, N_i\}$ . The letters  $I_i, D_i, Z_i, N_i$  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  $\sigma$  of operations of the finite control (i.e.  $c_{\sigma, j}$  is the difference  $|\sigma \uparrow I_j| - |\sigma \uparrow D_j|$  between the number of increments and decrements). We say that a word  $\sigma$  of the finite control is *halting* iff

- (1) for all prefixes  $\pi Z_j$  of  $\sigma$ ,  $c_{\pi, j} = 0$ , and
- (2) for all prefixes  $\pi N_j$  of  $\sigma$ ,  $c_{\pi, j} \neq 0$ .

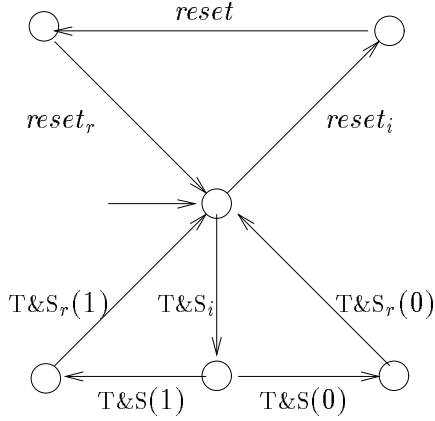


Figure 3: The language of invocations, commits, and responses for a Test&Set bit

joint alphabet  $\Sigma \cup \Sigma^{ir}$ , denoted by  $\Sigma^{oir}$ . As before, the subset of  $\Sigma^{oir}$  containing events by a single process  $p$  is denoted by  $\Sigma^{oir}(p)$ . For each process  $p$ , the language

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

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 Figure 3.

**Lemma 2** *A well-formed string  $\sigma$  over the alphabet  $\Sigma^{ir}$  is linearizable with respect to the specification  $S$  iff there exists a string  $\tau$  over the alphabet  $\Sigma^{oir}$  such that the following three constraints are satisfied:*

- (C1) *The projection  $\tau \upharpoonright \Sigma^{ir}$  equals  $\sigma$ .*
- (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  $S$ .* ■

Notice 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  $p$ ,  $\tau \upharpoonright \Sigma^{ir}(p)$  equals  $\sigma \upharpoonright \Sigma^{ir}(p)$  (i.e. every process sees the same sequence of invocations and responses). Thus, the string  $\sigma'$  of Figure 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:

$$\forall \square (invoke \rightarrow \exists \diamond response).$$

## 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. Sequences 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 of *transactions*. Every transaction  $T$  has a finite alphabet  $\Sigma_T$  of operations. We assume that for  $T \neq T'$ ,  $\Sigma_T$  and  $\Sigma_{T'}$  are disjoint. Each alphabet  $\Sigma_T$  includes two special letters  $begin(T)$  for Begin Transaction, and  $end(T)$  for End Transaction. Denote  $\Sigma'_T = \Sigma_T \setminus \{begin(T), end(T)\}$ . The set  $\Sigma = \cup_T \Sigma_T$  contains all events. The *specification* of a transaction  $T$  is the regular language  $S(T)$  which is required to be a subset of  $begin(T)(\Sigma'_T)^* end(T)$ .
- A finite set of *objects*. Every object  $A$  has a finite alphabet  $\Sigma_A$ . It holds that  $\cup_A \Sigma_A = \cup_T \Sigma'_T$ , i.e., every event besides begin and end transaction involves some object. The *specification* of an object  $A$  is the regular prefix closed language  $S(A)$ .

## 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. 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  $\Sigma^{ir}$ . 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  $\Sigma^{ir}$ . 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

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

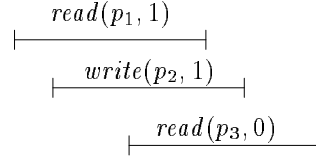
over the alphabet  $\Sigma^{ir}(p)$  is denoted by  $L^{ir}(p)$ . A string  $\sigma$  over the alphabet  $\Sigma^{ir}$  is *well-formed* iff for every process  $p$ ,  $\sigma \upharpoonright \Sigma^{ir}(p)$  is in the language  $L^{ir}(p)$ . A *concurrent implementation* is a language  $I$  consisting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

### Linearizability definition

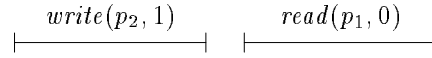
Recall that  $S$  is the asynchronous product of specifications of individual objects. Let  $S^{ir}$  be the language over the alphabet  $\Sigma^{ir}$  obtained by replacing every symbol  $o(p, A, w, v)$  by the string  $o_i(p, A, w)o_r(p, A, v)$ . Every string in  $S^{ir}$  is well-formed with responses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

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

Thus, for two events  $a, b \in \Sigma^{ir}$ ,  $(a, b)$  is in the dependency relation *lin* iff either both events belong to



Linearizable String  $\sigma$



Nonlinearizable, sequentially consistent string  $\sigma'$

Figure 2: Sample strings for atomic bit

the same process, or if the first event is a response, and the second one is an invocation. A well-formed string  $\sigma$  over the alphabet  $\Sigma^{ir}$  is *linearizable* with respect to the specification  $S$  iff  $\sigma$  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. For instance, for the atomic bit  $x$ , Figure 2 shows both a linearizable and non-linearizable string.

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  $I \subseteq cl_{lin}(S^{ir})$ .

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

**Lemma 1** *A well-formed string  $\sigma$  over  $\Sigma^{ir}$  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  $\sigma$  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  $\Sigma$  is in the specification language  $S$ . To formalize this intuition, consider

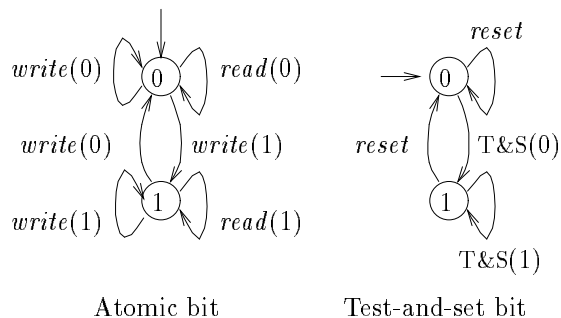


Figure 1: Specification of Test&Set and atomic bits

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

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  $S(A)$ .

## 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. For example, in the case of the atomic bit  $x$ , the event sequence

$$read(p, x, 0), write(p', x, 1)$$

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

$$write(p', x, 1), read(p, x, 0)$$

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.

Let  $\Sigma$  be the set of events of all objects. The specification  $S$  is the asynchronous product  $\parallel_A S(A)$ ; that is,

a string meets the sequential specification if its projection on individual objects satisfy their respective specifications. We say that a string  $\sigma$  is *sequentially consistent* iff there exists a string  $\sigma' \in S$  such that, for all processes  $p$ ,  $\sigma \upharpoonright p$  equals  $\sigma' \upharpoonright p$ . A sequentially consistent implementation is any language  $I$  over the events  $\Sigma$ , such that all strings in  $I$  are sequentially consistent.

An equivalent definition of this condition uses dependency relations. Define the (symmetric) dependency relation  $sc$  over  $\Sigma$  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 of different processes are commutable. By definition, a string  $\sigma$  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. We will find, however, that the problem of verifying that a finite state implementation is sequentially consistent, for a given set of concurrent objects, is undecidable. In the special case where only read and write operations are allowed, the problem remains undecidable (however, we leave open the possibility that an algorithm exists for some fixed number of objects less than 4). 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.



rithm [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.

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

#### Traces

A *concurrent alphabet* is a pair  $(\Sigma, D)$ , where  $\Sigma$  is a finite alphabet and  $D$  is a binary relation over  $\Sigma$  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  $(\Sigma, D)$ , define  $\Rightarrow_D$  to be the least relation over  $\Sigma^*$  satisfying

- (1)  $\Rightarrow_D$  is reflexive and transitive, and
- (2) for all words  $\sigma, \sigma' \in \Sigma^*$  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 word  $\sigma'$  can be obtained from  $\sigma$  by repeatedly commuting commutable pairs of letters. Given a concurrent alphabet  $(\Sigma, D)$

and language  $L$  over  $\Sigma$ , the *closure* of  $L$  with respect to  $D$ , denoted  $cl_D(L)$ , consists of all words  $\sigma'$  such that  $\sigma' \Rightarrow_D \sigma$  for some  $\sigma \in L$ .

For example, let  $\Sigma = \{a, b\}$  and  $L = (ab)^*$ . For  $D = \{(b, a)\}$ ,  $cl_D(L)$  contains all words  $\sigma$  such that  $\sigma$  contains the same number of  $a$ ’s and  $b$ ’s, and in every prefix of  $\sigma$ , the number of  $b$ ’s does not exceed the number of  $a$ ’s. Note the following facts regarding the closure operation:

1. If  $L$  is a context-sensitive language, then so is  $cl_D(L)$ .
2. It is possible that  $L$  is regular but  $cl_D(L)$  is not even 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 set  $W_o$  of input values for  $o$  and a set  $V_o$  of responses that  $o$  may return. Let  $P$  be a finite set of processes. The event  $o(p, A, w, v)$ , for an operation  $o \in O$ , a 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  $\Sigma(A)$ .

For 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 Figure 1.

Another example is a *test-and-set* bit with 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 Figure 1.

# Model-checking of correctness conditions for concurrent objects

Rajeev Alur  
Bell Laboratories  
alur@bell-labs.com

Ken McMillan  
Cadence Berkeley Labs  
mcmillan@cadence.com

Doron Peled  
Bell Laboratories  
doron@bell-labs.com

## Abstract

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 semi-commutative alphabet. The three model checking problems are shown to be, respectively, in PSPACE, in EXSPACE, and undecidable.

## 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 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 two transactions of different processes that do not overlap in time.

Each of these notions of correctness has its place. There are cases when simply 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<sup>1</sup>, this membership problem has a polynomial algo-

---

<sup>1</sup>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.