On Execution-Based Formalization for Sequential Consistency Verification

Ali Sezgin * Ganesh Gopalakrishnan

Abstract

The adequacy of execution based formalizations in modeling and verifying shared memory systems is discussed. It is argued that `unclear sentence...`. not comparable. We claim that the decidability problem of sequential consistency for finite-state systems is an open problem, in contrast with the result of [1].

Key words:

Shared memories, Sequential consistency, Formalization

PACS:

1 Introduction

The inception of shared memories as a way to increase the performance of computer systems has brought about its own problems. Checking whether a system behaves within a prescribed manner is one such problem. Testing, simulation and formal verification are different approaches to this problem. In this paper, we will be concerned with the latter.

* Corresponding author
Email addresses: `sezgin@cs.utah.edu` (Ali Sezgin), `ganesh@cs.utah.edu` (Ganesh Gopalakrishnan).

Formal verification here means that both the model and the system are represented as mathematical structures and the task is to prove that a certain relation holds between these structures. Formalization which is the transformation of real world entities into mathematical structures then is the first implicit step. Different formalizations might result in different mathematical structures not necessarily equivalent in expressiveness [1] . These non-equivalences in representations are likely to result in different properties for the structures which will in turn lead to different algorithms for verification.

Our objective in this paper is to critique execution-based, in particular trace theoretical, formalizations. We will argue that the absence of the concept of program in the formalization leads one to prove certain properties for the mathematical structures which do not necessarily hold for the real world entities they represent. We will also briefly talk about one major assumption which this formalization requires and which we think is no longer adequate for new systems.

2 Memory - An Intuitive Explanation

Let us briefly summarize the intuitive picture concerning memories. A memory is an information retaining device. It has a set of data values each of which is uniquely indexed with what is known as its address or location. A memory, within a system, communicates with other parties through receiving requests and generating *suitable* completions to these requests. To make matters simple, it is common practice to define only two classes of requests: reads and

[1] An obvious and well-known counter example to this assertion is the equivalence of Church's lambda-calculus and Turing machines, the formalizations of computation.

writes. A read request queries the content of an address. The suitable completion is the data value indexed by that address. A write request updates the content of an address. The suitable completion, in this case, can be a simple acknowledgment. In what follows, we will use the words *instruction* and *response* for requests and completions, respectively.

When the memory system receives its instructions from a single user, its correct mode of operation is easy to define: each read instruction of an address should return the value of the most recent written value to that address, or the initial value in the absence of such a write. The concept of "most recent" is well-defined in this context; in the absence of concurrency, there is a total temporal order of requests.

Things do get complicated, however, when there is more than one user which is the case for shared memory systems. It is generally accepted that enforcing the memory system to process instructions in the order they are dispatched by the users will result in poor performance of the system as a whole. Instead, a certain amount of non-determinism in the behavior of the memory system is allowed. Varying degrees of such non-determinism results in different correctness conditions for memory systems. These conditions are known as shared memory models.

Whatever the memory model is, one thing remains invariant: the (sequential) temporal order of requests per user is a parameter in the definition of the memory model. This temporal order is usually known as the program order, although it does not necessarily correspond to the sequential program order; whether it does depends on what the memory system represents. For a programmer, the memory system is the abstraction of the operating system,

the compiler, the hardware and finally the physical memory devices. For a designer, memory might abstract only the physical memory devices.

As an example for a shared memory model, consider the well-known model of sequential consistency [2]. It requires that the memory behaves as if it responds to a single user whose instruction stream is an interleaving of the instructions of all the users such that this interleaving respects the order of instructions per user. The non-determinism here lies in the fact that the interleaving does not necessarily correspond to the global temporal ordering of instructions among users; the memory system behaves according to sequential consistency as long as there is one interleaving that *logically explains* its behavior.

The next step in the verification process is to formalize the intuition summarized above. We will now describe one such formalization.

3 Execution-based Formalization

One way to formalize memory systems is to view them as machines generating responses. In this view, an *execution* of a memory system is the collection of responses, also called *events* in this framework, this memory system generates. A memory model is described in terms of a *model predicate* over executions. A memory system satisfies a memory model if all the executions the system generates satisfies the model predicate.

Let us first start with the formalization of the memory system. We will parameterize a memory system over the set of users, the set of addresses and the set of different data values each address can hold, represented by P, A and D, respectively. We will take all these sets as finite. We will also assume that,

for the rest of this paper, these sets are fixed.

A read event is represented by $r(p, a, d)$ where $p \in P$ is the processor that issued the instruction, $a \in A$ is the address queried by the read instruction and $d \in D$ is the data value returned by the memory. Similarly, a write event is represented by $w(p, a, d)$ with p, a, and d having the same meanings. Σ is the alphabet containing all read and write events.

The parameters of a read (write) event are extracted using the functions π, α and δ. That is, for $s = r(p, a, d)$, $\pi(s) = p$, $\alpha(s) = a$ and $\delta(s) = d$.

How an execution is represented results in different formalizations. There have been research that used partial orders [3], graphs [4,5] and traces [1,6–8]. We will consider the latter which has almost always been used in the verification of sequential consistency.

In trace-theoretical representation, we use a partially commutative monoid instead of the free monoid Σ^*. Let σ_1, σ_2 be strings over Σ, let s, t be symbols in Σ and let $\sigma = \sigma_1 st \sigma_2$. Then the string $\sigma_1 ts \sigma_2$ is 1-step equivalent to σ if $\pi(s) \neq \pi(t)$. An equivalence class is the transitive closure of 1-step equivalence. We can say that two strings not necessarily syntactically equal but belonging to the same equivalence class have the same semantic value.

A string $\sigma = s_1 s_2 \cdots s_n$ for $s_i \in \Sigma$ is *serial* if for any $i \leq n$ such that $s_i = r(p, a, d)$ is a read event, either there exists $j < i$ with $\alpha(s_j) = a$, $\delta(s_j) = d$, and there does not exist $j < k < i$ such that $\alpha(s_k) = a$ and $\delta(s_k) \neq d$, or d is the initial value of a. For simplicity, we will assume that the initial value for each address is 0.

In this formalization, an execution is a string over Σ. The model predicate for

sequential consistency is as follows: An execution is sequentially consistent if it is in the equivalence class of a serial string. We say that a memory system is sequentially consistent if all its executions are sequentially consistent.

Based on this formalization, it has been claimed that [1] a sequentially consistent finite-state memory system has a sequentially consistent regular language. Consequently, in [1], it is proved that it is undecidable to check for an arbitrary finite-state memory system whether it is sequentially consistent or not. This result has been cited in almost all of the subsequent work such as [7,8,5,9]. Before arguing the relevance of this result, however, it first behooves us to talk about an assumption that has not been explicitly stated.

4 Trace-based Formalization and In-order Completion

We have said that the definition of sequential consistency, or any memory model for that matter, required information on the sequential order of instructions issued per processor, also known as the program order. On the other hand, we have not really talked about program order in the context of trace-based formalization. The conciliation of these two seemingly contradicting facts lies in a crucial assumption: the memory system is expected to complete the requests it receives in an order which respects per processor issuing order. That is, if the memory system receives instruction i_1 at time t_1 from processor p, instruction i_2 at t_2 again from the same processor and $t_1 < t_2$, then it is assumed that i_1 completes[2] before i_2. That is precisely why the equivalence classes defined above do respect program order; events

[2] This notion might also be called "commitment".

belonging to the same processor are not allowed to commute, hence at each 1-step equivalence the program orders remain the same.

It is highly questionable whether this assumption can stay valid, given the ever ambitious optimizations done for memory systems. There are already memory systems which process their requests out of issuing order.

Consider the following scenario. Processor p issues $r(p,a)$ [3] and then issues $r(p,b)$. If the second read completes before the first one, what we observe in the execution will be of the form $\sigma_1 r(p,b,d)\sigma_2 r(p,a,d')\sigma_3$ for strings σ_i over Σ. Any string in the equivalence class of this string will always have $r(p,b,d)$ before $r(p,a,d')$, contradicting the initial program order.

One can say that an intermediate machine that would convert what the memory system generates into a string for which the assumption holds can be constructed. We could then take the combination of the memory system and that machine and work on the output of the intermediate machine without any problem. However, there are cases where a finite-state machine simply cannot generate such an output.

Consider now a slight variation of the above scenario. Processor p issues $r(p,a)$ and then issues an unbounded number of $r(p,b)$. That is, after reading address a, it polls the address b for an unbounded number of times. Assume further that the read of a does not return a value unless all the reads of b complete. This will mean that the finite-state intermediate machine must have the capability of storing an unbounded amount of information, in this case all the

[3] This is the representation of the instruction whose response is the read event $r(p,a,d)$ for some $d \in D$.

read events of address b. This is clearly and theoretically impossible.

This assumption of in-order completion found in trace-based formalization, therefore, restricts its use to a subset of all possible memory systems, not all of which are pure theoretical concoctions.

Unfortunately, not only all possible finite-state memory systems cannot be formalized using trace theory, the finiteness of a memory system represented in this formalization cannot be formulated either. We will argue this point next.

5 Finiteness and Trace-based Formalization

It has been argued in [1] that since a memory system is basically a finite-state automaton whose language is a subset of Σ^*, the memory system is finite-state if and only if its language is regular. Furthermore, as we have previously mentioned, this implies that a finite-state memory system is sequentially consistent if and only if its language is regular and sequentially consistent.

However, we believe that this characterization of finiteness is inadequate. Consider the following set of executions, given as a regular expression:

$$w(1,a,2)r(1,a,1)^*r(2,a,2)^*w(2,a,1)$$

According to the definition of sequential consistency, the memory system generating this language is sequentially consistent. It is sequentially consistent because any string belonging to this regular expression has a serial string in

its equivalence class. For instance, the execution

$$w(1,a,2)r(1,a,1)r(2,a,2)w(2,a,1)$$

is equivalent to the serial string

$$w(1,a,2)r(2,a,2)w(2,a,1)r(1,a,1)$$

Let us assume that N is the cardinality of the state space of the finite-state memory system generating this regular expression. Think of the execution where we have $2N$ $r(1,a,1)$ events and $2N$ $r(2,a,2)$ events. By the execution string, we know that the first event is $w(1,a,2)$. This is to be followed by the read event $r(1,a,1)$. Note that, by the assumption discussed in the previous section, we know that, without any information about the relative issuing orders among read instructions belonging to different processors, at least $2N$ instructions must be issued by the second processor before the write instruction which is the last to be committed is issued by this same processor.

However, this cannot be done by a sequentially consistent and finite-state machine. Noting that the cardinality of the state space of the machine was N, there are two possibilities:

(1) The machine generates the read event $r(1,a,1)$ before the issuing of the instruction corresponding to the event $w(2,a,1)$. If at the instant the machine generates this read event we stop feeding the finite-state machine with instructions, it will either terminate with an execution that does not have a serial string in its equivalence class or it will hang waiting for issuing of the write instruction it *guessed*. The former case results in a non-sequentially consistent execution. The latter case where the memory

system simply refuses to terminate computation will be discussed below.

(2) The machine generates the first read event after the issuing of the instruction corresponding to the $w(2, a, 1)$ event. This means that the machine has not generated any event for at least $2N$ steps. This in turn implies that, since there are N states, there exists at least one state, s, which was visited more than once, such that on one path from s to s, the machine inputs instructions but does not generate any events. Let us assume that the mentioned path from s to s was taken k times. Consider a different computation where this path is taken $2k$ times; each time this path is taken in the original computation, in the modified computation it is taken twice. It is not difficult to see that this will change the program, the number of instructions issued, but will leave the execution the same; no output is generated on the path from s to s. Hence, we obtain an execution which does not match its program; the program's size becomes larger than the size of execution. Put in other words, the finite-state memory ignores certain instructions and does not generate responses.

The basic fallacy here is the abstraction of input, or the program. In the first case, where the memory system hangs or does not terminate, the memory system cannot be considered correct in any reasonable sense. A memory system should always generate an execution as long as the stream of memory accesses, or instructions, are syntactically correct.

In the second case, we have a memory system which generates its output regardless of what it receives as input. There should be a well-defined correspondence between the instructions a memory system receives and the responses it generates.

Remember that the initial motivation for shared memory models was to capture some sort of correctness for shared memory systems. The two *ground rules* we have mentioned above, that the memory system does not deadlock and that the program and its execution must be related, should be properties that are satisfied by any memory system, not only sequentially consistent systems. However, it is impossible to characterize these requirements when only execution is present in the formalization.

If the reader is not convinced about the necessity of ground rules, we could propose an alternative argument. Going back to the original definition, we note that a sequentially consistent memory system is required to behave as if it is a single user system. A single user memory system, on the other hand, cannot exhibit any of the behaviors mentioned above (deadlock or arbitrary execution) and be deemed correct.

It is therefore not correct to prove a property in trace-based formalization and then claim that property to hold for memory systems in general. The reverse direction holds as well: certain properties of memory systems cannot be expressed in trace-based formalization. Finiteness is one such property. We have been so far unable to characterize for the trace-based formalization the set of executions which can be generated by finite-state memory systems.

Another property that has been proved to hold for memory systems in trace-based formalization is the undecidability we mentioned above. As a corollary of the argument we have given for the finiteness, the result of undecidability is not applicable to finite memory systems in general. We claim that the decidability of checking the sequential consistency of a finite-state memory system is still an open problem.

It is worth noting that these observations have led us to develop a novel formalization framework for shared memories[10]. In the formalization framework we propose, both the program and the execution are part of the definition of a shared memory system as well as that of a shared memory model. For instance, finiteness is captured easily by requiring the relation realized by the memory system to be rational!

References

[1] R. Alur, K. McMillan, D. Peled, Model-checking of correctness conditions for concurrent objects, in: Symposium on Logic in Computer Science, IEEE, 1996, pp. 219–228.

[2] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, ACM Transactions on computer 28 (9) (1979) 690–691.

[3] P. Kohli, G. Neiger, M. Ahamad, A characterization of scalable shared memories, Tech. Rep. GIT-CC-93/04, College of Computing, Georgia Institute of Technology (January 1993).

[4] K. Gharachorloo, Memory consistency models for shared-memory multiprocessors, Ph.D. thesis, Stanford University (December 1995).

[5] R. Nalumasu, Design and verification methods for shared memory systems, Ph.D. thesis, Department of Computer Science, University of Utah (1999).

[6] J. D. Bingham, A. Condon, A. J. Hu, Toward a decidable notion of sequential consistency, in: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures, ACM Press, 2003, pp. 304–313.

[7] A. E. Condon, A. J. Hu, Automatable verification of sequential consistency, in:

13th Symposium on Parallel Algorithms and Architectures, ACM, 2001, pp. 113–121.

[8] T. A. Henzinger, S. Qadeer, S. K. Rajamani, Verifying sequential consistency on shared-memory multiprocessor systems, in: Proceedings of the 11th International Conference on Computer-aided Verification (CAV), no. 1633 in Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 301–315.

[9] S. Qadeer, Verifying sequential consistency on shared-memory multiprocessors by model checking, Tech. Rep. Research Report 176, Compaq SRC (December 2001).

[10] A. Sezgin, Formalization and verification of shared memory, Ph.D. thesis, School of Computing, University of Utah (2004).