On the Decidability of Shared Memory Consistency Verification

Ali Sezgin
Department of Computer Engineering
Atılım University
Gölbaşı, 06836 Ankara
Turkey
asezgin@atilim.edu.tr

Ganesh Gopalakrishnan*
School of Computing
University of Utah
Salt Lake City, UT 84108
USA
ganesh@cs.utah.edu

Abstract

We view shared memories as structures which define relations over the set of programs and their executions. An implementation is modeled by a transducer, where the relation it realizes is its language. This approach allows us to cast shared memory verification as language inclusion. We show that a specification can be approximated by an infinite hierarchy of finite-state transducers, called the memory model machines. Also, checking whether an execution is generated by a sequentially consistent memory is approached through a constraint satisfaction formulation. It is proved that if a memory implementation generates a non interleaved sequential and unambiguous execution, it necessarily generates one such execution of bounded size. Our paper summarizes the key results from the first author's dissertation, and may help a practitioner understand with clarity what "sequential consistency checking is undecidable" means.

1. Introduction

Shared memory consistency models ("consistency models") are centrally important in the design of high performance hardware based on shared memory multiprocessing (e.g., [1]) as well as high performance software based on shared memory multi-threading (e.g., [2]). To mitigate the complexity of designing shared memory consistency protocols, either post-facto verification (e.g., [3]) or correct by construction synthesis (e.g., [4]) are employed. Shared memory multiprocessor machines are programmed according to their consistency models, which define the possible outcomes of running concurrent programs. The semantics of shared memory are described by their consistency models which specify the set of values that `loads` are permitted to return for any program. Typical programs consist of `loads`, `stores`, and other special instructions such as `barriers`, and `fences`. In this paper, we consider only `loads` (synonymous with `reads`) and `stores` (synonymous with `writes`), as is customary in a study of basic shared memory consistency issues.

For most practical purposes, weak shared memory models such as the Sparc TSO [5] or the Itanium memory model [6] are of interest. However, most programmers understand shared memory in terms of *sequential consistency* (SC) for three reasons: (i) SC has, traditionally, been the memory model of choice to support in hardware; (ii) excellent theoretical understanding exists with respect to SC; and (iii) programmers strive to obtain a semantics that matches SC for particular programs of interest by inserting the least number of fences [7, 8]. Thus, it is important to have theoretical issues about sequential consistency well understood by programmers in simple and intuitive terms, and in terms of models that they can easily relate to. We believe that this is not the case today: there are results which can be misunderstood, practical issues that have not been considered adequately, and in addition, new results that warrant dissemination at an intuitive level. The aim of this paper is to offer such a perspective to practitioners as well as to those in formal methods.

As an example of what we mean, consider [9] in which the authors have shown that the problem of verifying finite-state shared memory consistency protocols [10] against the sequential consistency memory model is undecidable. Upon closer examination, [9] does *not* offer a definite verdict on the practical aspect of shared memory consistency protocol verification. What [9] show is that *if* a shared memory system is viewed in terms of *traces* of executed instructions, then the problem of showing that these traces are contained in the language of sequential consistency (formally defined in [9]) is undecidable. In [11], we show that if we model a finite-state consistency protocol in terms of triples of the form

*Supported in part by NSF grant ITR-0219805 and SRC Contract 1031.001

$\langle program, execution, correspondence \rangle$, where we not only keep the executions (as in current approaches) but also (i) model the programs that gave rise to the executions, and (ii) model a correspondence relation between which program instructions appear where in the execution, then the question of decidability remains open. We argue that our way of modeling memory systems avoids all the problems pertaining to realizability that a trace based view invites. Moreover, we believe that our model is much more faithful to reality in that a shared memory system is a *transducer* from programs to executions and not merely an autonomous process that spews executed instructions.

As another example of a less known result, we recently show [11] that for *unambiguous* executions (executions where each address a is not written into with the same value more than once), the question of verifying SC becomes decidable. The manner in which this result was obtained is, in itself, interesting. We show that given any unambiguous execution, one can generate constraints that capture orderings that must be *disobeyed* for the execution to be sequentially consistent. We show that these constraints imply a bound on the size of executions to be searched. This result has been obtained without making any assumptions such as *location monotonicity* or *symmetry* that are made in [12, 13]. It is also the first time that we believe that the notion of decidability and ambiguity have been related.

Views similar to ours exist in other work also: for example, in [14], the authors point out that the class of protocols considered by [9] possibly include instances that are irrelevant (unrealizable) in practice. They go on to provide a characterization of decidable families of protocols, as well as decision procedures. Others [15] have also pointed out such decidable protocol classes. However, these decidable SC characterizations leave out important classes of executions that our approach considers.

Roadmap. In Section 2, we define the notion of a memory model as a relation over programs and executions and a shared memory system as a transducer. We address many details that are not addressed in related works (e.g., [9, 14]) without which the connections between definitions and physically realizable systems are unclear; these include notions such as (i) establishing a relation between memory requests and responses using a *coloring relation*, (ii) the notions of *immediate* and *tabular* that allow a finite-state protocol to interact with a (potentially) out-of-order memory system and collect responses corresponding to requests. In Section 3, we describe what happens if we certify a memory system to be correct solely based on executions (without considering programs). In Section 4, we describe a finite approximation to sequentially consistent shared memory systems. In Section 5, we present a constraint based approach to verify finite executions, and state decidability results applicable to unambiguous executions. While we cannot do justice to the level of detail it takes to explain these notions adequately (which is what [11, 16] do), what we hope to achieve is an intuitive dissemination of our results which our former publications do not do.

2. Formalization of Shared Memory

Any work on shared memory formalization or verification has to start with an understanding of what a memory entails. After all, shared memory is but a type of memory. It is common practice to immediately start with a mathematical definition. Here, we will start with an intuitive explanation, stating the obvious, and build our formalization on top of that.

We will explain a memory system using two orthogonal and complementary views: static and dynamic. Statically, a memory system is an array whose dimension is known as the *address space*. What each address can hold as datum forms the *data space* of the memory. Dynamically, a memory system is an interacting component which generates response to each instruction it receives. The instructions it receives are broadly classified as those that query and those that update. The instructions belonging to the former class are usually called *read* instructions; those in the latter class are called *write* instructions. The state of a memory system can be uniquely defined as the combination of the contents of its address space (static part) and the set of instructions it is processing (dynamic part).

What distinguishes a shared memory from other types of memory is the environment with which the memory system interacts. In a shared memory, typically, there are several users and each instruction is tagged with the identifier of the user issuing the instruction. Hence, contrary to a single user system, not only does the memory differentiate an instruction according to its class or the address on which it operates, but also according to its issuer.

A shared memory system has multiple users and as such it forms a concurrent system. Removing this concurrency at the memory side goes against the raison d'être of shared memory systems, i.e. increased performance through parallelism. Allowing arbitrary behavior by the memory system would make programming infeasible. The middle ground is to define a set of behaviors: each instruction sequence (*program*) is expected to result in one of the possibly several allowed response sequences (*executions*). A *shared memory model* defines this relation. When a shared memory system is claimed to conform to a certain shared memory model, it is to be understood that a program can only result in an execution defined by this shared memory model. Formal verification, then, is to prove this *claim*.

We keep referring to two seemingly different notions: a shared memory model and a shared memory system. This is not arbitrary. A shared memory model should be the definition of a relation (what it contains) and not its description (how it realizes). A shared memory system, on the other hand, should be the formal model of a design. It should describe how it is to behave for each program it receives.

In our framework, closely following intuition, a shared memory model is a binary relation over programs and executions, called a *specification*. A specification is parameterized over the set of users, address space and data space. The instructions or the responses that the memory might receive or generate and which response can be generated for which instruction forms a structure called *interface* and is also part of the specification.

Definition 1 *A memory interface, $\mathfrak{F}$, is a tuple $\langle \mathcal{I}, \mathcal{O}, \rho \rangle$, where*

1. *$\mathcal{I}$ and $\mathcal{O}$ are two disjoint, nonempty sets, called input (instruction) and output (response) alphabets, respectively. Their union, denoted by Σ, is called the alphabet.*

2. *$\rho \subseteq \mathcal{O} \times \mathcal{I}$ is the response relation.*

Definition 2 *The rw-interface is the memory interface $\mathcal{RW}$ with (here $\mathbb{N}$ is the set of natural numbers):*

1. *$\mathcal{I}^{\mathcal{RW}} = \{\mathtt{w_i}\} \times \mathbb{N}^3 \cup \{\mathtt{r_i}\} \times \mathbb{N}^2$*

2. *$\mathcal{O}^{\mathcal{RW}} = \{\mathtt{w_o}, \mathtt{r_o}\} \times \mathbb{N}^3$*

3. *For any $\sigma_i \in \mathcal{I}^{\mathcal{RW}}$, $\sigma_o \in \mathcal{O}^{\mathcal{RW}}$, we have $(\sigma_o, \sigma_i) \in \rho^{\mathcal{RW}}$ iff either the first component of σ_o is $\mathtt{w_o}$, the first component of σ_i is $\mathtt{w_i}$ and they agree on the remaining three components, or the first component of σ_o is $\mathtt{r_o}$, the first component of σ_i is $\mathtt{r_i}$ and they agree on the second and third components. Formally,*

$$\begin{aligned}\rho^{\mathcal{RW}} = &\{((\mathtt{w_o}, p, a, d), (\mathtt{w_i}, p, a, d)) \mid p, a, d \in \mathbb{N}\} \cup \\ &\{((\mathtt{r_o}, p, a, d), (\mathtt{r_i}, p, a)) \mid p, a, d \in \mathbb{N}\}\end{aligned}$$

Also, for ease of notation the following will be used:

1. *A partition of Σ, $\{R, W\}$, where*

$$\begin{aligned}R &= \{\mathtt{r_o}\} \times \mathbb{N}^3 \cup \{\mathtt{r_i}\} \times \mathbb{N}^2 \\ W &= \{\mathtt{w_i}, \mathtt{w_o}\} \times \mathbb{N}^3\end{aligned}$$

2. *Three functions, π, α, δ, where for any $\sigma \in \Sigma^{\mathcal{RW}}$, $\pi(\sigma)$ is the value of σ's second component, $\alpha(\sigma)$ that of the third component, and $\delta(\sigma)$ that of the fourth component if it exists, undefined (denoted by $\perp$) otherwise.*

Definition 3 *A memory specification, $\mathfrak{S}$, for a memory interface $\mathfrak{F}$ is the tuple $\langle \mathfrak{F}, \lambda \rangle$, where $\lambda \subseteq ((\mathcal{I}^{\mathfrak{F}})^* \times (\mathcal{O}^{\mathfrak{F}})^*) \times$ $\mathbf{Perm}$, is the input-output relation.*

Here, $\mathbf{Perm}$ is the set of all permutations. We later employ $\mathbf{Perm}_k$ for the set of all permutations over $\{1 \ldots k\}$. We shall let $\mu^{\mathfrak{S}}$ denote $dom(\lambda^{\mathfrak{S}})$ (a relation over $(\mathcal{I}^{\mathfrak{S}})^* \times (\mathcal{O}^{\mathfrak{S}})^*$). λ of a memory is expected to define the relation between the input to a memory, a (finite) string over $\mathcal{I}$ which might be called a *program* or an *instruction stream*, and the output it generates for this input, a (finite) string over $\mathcal{O}$ which might be called an *execution* or a *response stream*.[1] For each such program/execution pair of the memory, λ also defines, through permutation, the mapping between an individual instruction of the program and its corresponding output symbol in the execution.[2]

For instance, consider an input-output relation for $\mathcal{RW}$ which has the following element: $((((\mathtt{r_i,1,1})\ (\mathtt{r_i,1,1})),((\mathtt{r_o,1,1,2})\ (\mathtt{r_o,1,1,4}))),(21))$. In the program, we have two reads issued by processor 1 to address 1. The execution generates two different values read for address 1; 2 and 4. By examining the permutation, we see that the first instruction's response is placed at the second position of the output stream, whereby we conclude that the returned value for the first read is 4. Similarly, the second read's value is 2. So, intuitively, if the permutation's i^{th} value is j, the j^{th} symbol of the output stream is the response corresponding to the i^{th} instruction of the input stream.

Definition 4 *A memory specification $\mathfrak{S}$ is called proper if*

1. *$\mu^{\mathfrak{S}}$ is length preserving.*

2. *For any $\mathbf{p} \in (\mathcal{I}^{\mathfrak{S}})^*$, there exists $\mathbf{q} \in (\mathcal{O}^{\mathfrak{S}})^*$ such that $(\mathbf{p}, \mathbf{q}) \in \mu^{\mathfrak{S}}$.*

3. *$\sigma = (\mathbf{p}, \mathbf{q}) \in \mu^{\mathfrak{S}}$ implies $\emptyset \neq \lambda^{\mathfrak{S}}(\sigma) \subseteq \mathbf{Perm}_{|\mathbf{p}|}$ and for any $\eta \in \lambda^{\mathfrak{S}}(\sigma)$, $\eta(j) = k$ implies $\rho^{\mathfrak{S}}(q_k, p_j)$.*

If the first condition holds, the memory specification is *length-preserving*. Then, a length-preserving memory specification is one which matches the length of its input to its output. Note that, without the third requirement, it is not of much use. **Example:** $\mathcal{S}^{ND} = \langle \mathcal{RW}, \lambda^{ND} \rangle$, where $\sigma = ((\mathbf{p}, \mathbf{q}), \mathbf{n}) \in \lambda^{ND}$ implies $\mathbf{p} \in (\mathcal{I}^{\mathcal{RW}})^*$, $\mathbf{q} \in (\mathcal{O}^{\mathcal{RW}})^*$, $|\mathbf{p}| = |\mathbf{q}|$, $\rho^{\mathcal{RW}}(q_j, p_j)$ and $\eta(j) = j$, for $j \in [|\mathbf{p}|]$, $\eta \sim \mathbf{n}$ (η is the permutation represented by $\mathbf{n}$). The shared memory $\mathcal{S}^{ND}$ is length-preserving. If the second condition holds, a memory specification is *complete* (e.g., $\mathcal{S}^{ND}$ is complete). Completeness is the requirement that a memory specification should not be able to reject any program

[1] Although we are using the words *program* and *execution*, we do not claim that the input is required to be the unfolding of a program and the output to be its associated execution. This might or might not be the case, depending on where exactly the interface, user and memory are defined. One choice might put the compiler at the user side, quite possibly resulting in an input stream that is different from the actual ordering of instructions in a program due to performance optimizations.

[2] By itself, ρ defines the *type* of response relations allowed.

as long as it is syntactically correct with respect to the interface. This property, despite its simplicity, is one which has been neglected by all previous work on shared memory formalization, to the best of our knowledge ([17] considers some of these issues). The third condition is saying that any permutation used as a mapping from the instructions of the input to the responses of the output should be respecting the response relation of the interface. There are some subtle points to note. First, it requires that the length of the output stream, $|\mathbf{q}|$, to be at least as much as the length of the input stream, $|\mathbf{p}|$; it could be greater (a problem which is taken care of by the requirement of length-preserving). Second, even for the same input/output pair, there can be more than one permutation. Since we are trying to define a correct specification without any assumptions, these seemingly not tight enough requirements are favored for the sake of generality. $\mathcal{S}^{ND}$ satisfies this third property.

Consistency models are viewed as sets of triples

$$\langle program\_string, execution\_string, permutation\rangle$$

where the permutation describes the association between the individual instructions in the program string and the corresponding "finished" (or executed) elements in the execution string. For example, the triple `<Prog: write(p1,a,2); read(p2,a), Exec: read(p2,a,0); write(p1,a,2), Perm: 21>`

(where permutation `21` is an abbreviation for $\{(1,2),(2,1)\}$ – focussing only on the range elements), could be one element in the set that defines sequential consistency. Note that the standard notion of "program order" can be extracted from the first element of the triple (the program string) by projecting the string to individual processors.

The execution strings, which represent the temporal order, can be transformed, possibly into a different structure, such as a poset, such that the predicate of the consistency model is satisfied. For sequential consistency, however, a poset structure is not needed; one can transform the execution strings to a *serial* logical order, which is a total order as described earlier.

Consistency protocols are viewed as finite-state machines over finite strings. The alphabet of these machines consists of instructions paired with colors. The colors serve as a "marker dye". We color an instruction (e.g., `<write_i(p,a,d),blue>` or `<read_i(p,a),green>`) when it enters the system (also notice our use of the `_i` subscript to denote the issuing event corresponding to these instructions). When the instruction emerges after having being executed, we can tell by observing its color which program instruction gave rise to it (and also we mark the completion event of these instructions by the `_o` subscript).

We state well-formedness conditions for specifications and implementations. Some of these conditions are:

- The specification and implementation effect length-preserving maps from programs to executions.
- The implementation cannot accumulate an arbitrary number of instructions that it has ingested.
- The color sets are finite. This models the fact that in any finite-state implementation of a consistency protocol, the number of outstanding (unfinished) memory instructions is bounded.
- The implementation makes a "color association" between input symbols and output symbols that does not change as more instructions are considered. This captures that the association is decided by a deterministic process carried out by a finite-state protocol (we call this property *immediate*).
- The color association is defined by *pending* instructions alone. In other words, "finished input instruction / output response" pairs have no effect in deciding the nature of the color association for future instructions (we call this property *tabular*).

There is one additional and important problem: the mapping between instructions and their associated responses. The usual solution is to impose certain restrictions on the memory system such as in-order completion. For instance, if two read instructions of the same address by the same user await responses, the first suitable generated response (same address and user) belongs to the instruction that was issued first. We feel that this is an unnatural restriction and cannot be reasonably enforced on all memory systems.

Had we been dealing with infinite state machines, the solution would have been trivial: mark each instruction with a unique natural number and tag its response with the same number. This is, in fact, employed in defining specifications as we saw above. For finite-state systems, an infinite alphabet is not possible. Instead, we will let these machines have alphabets where each instruction and response is paired with *colors*. These colors will serve as a "marker dye". We color an instruction (e.g., $\langle ?a_2, \texttt{blue}\rangle$) when it enters the system. When a (colored) response emerges from the memory system, we can tell from its color which instruction gave rise to it.

In the most general case, a function has to be supplied to interpret pairs of strings over colors: given any pair of strings of equal length, this function would generate a permutation which would map instructions to responses. A color set together with such a (*conversion*) function is called a *coloring scheme*. it is not hard to see that this might result in syntactically different, semantically equivalent strings, something we are trying to avoid. Fortunately, we can do better. In order to justify the use of a canonical coloring, we allude to finitary arguments. When a user issues an instruction, it must have a certain mechanism to tell which response it receives actually corresponds to that instruction, especially when both the user and the memory system operate in a setting where out of order execution and pending instructions, instructions that have not yet received a response from the memory system, are allowed. Let us assume that i is an instruction that the user issued and the response r is the symbol that the memory system generated for i. When the user receives r from the memory system, it should be able to match it with i without waiting for other responses. Furthermore, once i and r are paired by the user, they should remain so; a future sequence of instructions and responses should not alter the once committed matchings. Since the user is a finite-state entity, it can retain only a finite amount of information about past input; most likely, it will only keep track of the *pending* instructions. These ideas are the basis for requiring implementations to be *immediate* and *tabular*[11].

Once an implementation is assumed to be immediate and tabular, and this assumption only depends on the finiteness of the system and the users, we can do away with arbitrary colorings and work with a canonical coloring. We have proved the existence of an equivalent canonical coloring for an arbitrary coloring in [11]. This means that any shared memory system can be modeled by a transducer which uses the canonical coloring.

3. Execution-based Formalism

An alternative, and widely adopted, 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.

As usual, a memory system is parameterized 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. 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 [18], graphs [19, 20] and traces [9, 14, 21, 22]. 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* (interleaved-sequential) 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. This is the standard definition for sequential consistency; it requires that each execution allow a (logical) reordering such that any read of an address returns the value of the most recent write to the same address.

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 [9] a sequentially consistent finite-state memory system has a sequentially consistent regular language. Consequently, in [9], 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 [21, 22, 20, 23]. Before arguing the relevance of this result, however, it first behooves us to talk about an assumption that has not been explicitly stated.

3.1. 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[3] before i_2. That is precisely

[3] This notion might also be called “commitment”.

why the equivalence classes defined above do respect program order; events 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 $r(p, a)$[4] 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 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 in this formalization cannot be formulated either. We will argue this point next.

3.2. Finiteness and Trace-based Formalization

It has been argued in [9] 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

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

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 *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 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.*

4. Finite Approximations to Sequential Consistency

In this section, we will define for each shared memory instance a set of machines whose language-union will cover all possible interleaved-sequential program/execution pairs of that instance at the initial state ι.

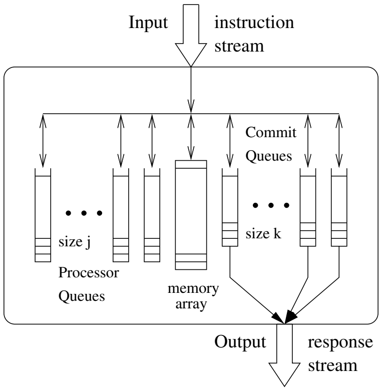

Figure 1. The diagram of $SC_{\mathcal{P},C}(j,k)$

Let $\mathcal{P}$ be a parameterized instance (P, A, D), C be a color set and let $j, k \in \mathbb{N}$. For simplicity, we will assume that $P = [|P|]^5$, $A = [|A|]$, $C = [|C|]$. Then, the machine $SC_{(\mathcal{P},C)}(j,k)$ is defined as follows:

There are $|P|$ *processor* fifo queues each of size j such that each queue is uniquely identified by a number in P, $|C|$ *commit* fifo queues each of size k, again each with a unique identifier from C, and the *memory array*, mem, of size $|A|$. Initially, the queues are empty, and the memory array agrees with ι, that is, $mem(i) = \iota(i)$, for all $i \in dom(\iota)$. At each step of computation, the machine can perform one of the following operations: read an instruction, commit an instruction or generate a response. The choice is done nondeterministically among those operations whose guards are satisfied.

Let $\sigma = (p, c)$ be the first unread instruction. The guard for reading such an instruction is that the $\pi(p)^{th}$ processor queue and the c^{th} commit queue are not full. If this operation is chosen by the machine, then one copy of σ is inserted to the end of the $\pi(p)^{th}$ processor queue, another is inserted to the end of the c^{th} commit queue and a link is established between the two entries.

The guard for committing an instruction is the existence of at least one nonempty processor fifo queue. If this guard

[5] For a set A, $|A|$ gives the cardinality of A. For a natural number n, [n] gives the set $\{0, 1, \ldots, n-1\}$.

is satisfied and the commit operation is chosen, then the head of one of the nonempty processor queues is removed from its queue. Let us denote that entry by (q, c). If $q \in R$, then the response $((\mathtt{r}_\circ, \pi(q), \alpha(q), mem(\alpha(q))), c)$ replaces the entry linked to (q, c) in the c^{th} commit queue. If $q \in W$, then the response $((\mathtt{w}_\circ, \pi(q), \alpha(q), \delta(q)), c)$ replaces the entry linked to (q, c) in the c^{th} commit queue and the $\alpha(q)^{th}$ entry of the memory array is updated to the new value $\delta(q)$, i.e., $mem[\alpha(q)] = \delta(q)$.

The guard for outputting a response is the existence of at least one nonempty commit queue which has a completed response at its head position. If indeed there are such nonempty queues and the output operation is chosen, then one of these commit queues is selected randomly, its head entry is output by the machine and removed from the commit queue.

Let the language of an $SC_{\mathcal{P},C}(j, k)$ machine, $L(SC_{\mathcal{P},C}(j, k))$, be the set of pairs of input accepted by the machine and output generated in response to that input. Let $L_{\mathcal{P},C}$ denote the (infinite) union $\bigcup_{j,k \in Nat} L(SC_{\mathcal{P},C}(j, k))$. In [11], we prove that any program/execution pair is interleaved-sequential only if it can be generated by some SC machine. This implies that $L_{\mathcal{P},C}$ contains all and only interleaved-sequential executions; that is, it is equivalent to the set of all sequentially consistent program/execution pairs.

The relation realized by a finite $SC_{\mathcal{P},C}(j, k)$ is also the language of a 2-tape automaton, since it is finite-state and length preserving (see [24]). The same can be said about length-preserving shared memory implementations of a finite instance. Since the emptiness problem for regular languages is decidable, it follows that it is decidable to check whether a finite instance implementation realizes a relation that is included in the language of some SC machine. Furthermore, completeness of an implementation of a finite instance is also decidable; it suffices to construct a new automaton with the same components whose transition labels are projected to the first (input) alphabet and then to check for its universality. These observations allow us to claim that it is decidable to check whether a memory system M is complete and has a language that is subset of $SC_{\mathcal{P},C^M}(j, k)$, for some $j, k \in \mathbb{N}$. Note that SC machines allow a semi-decision procedure for sequential consistency conformance of a protocol to be obtained through language containment (since we do not know how to bound j and k yet, a decision procedure is not obtained).

As a case study for the above ideas, in [11], we prove finite instances of lazy caching [25] sequentially consistent. The method we used is based on (regular) language inclusion and, thus, in principle, could be fully automated.

5. A Constraint Satisfaction Approach

We said that a concurrent execution is a combination of sequential executions, one per processor and the concurrent execution is interleaved-sequential if a certain interleaving of the sequential executions appears as if executed by a single processor. Let us call this the *logical order* of a concurrent execution. The logical order, then, is a fictitious order that conforms to all the requirements enforced by each processor. But what exactly do we mean by these requirements?

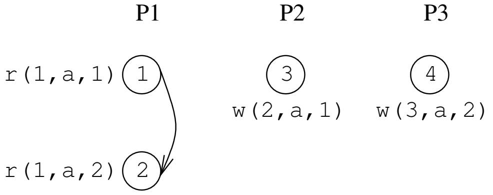

Figure 2. Sample concurrent execution, G_1.

Look at the concurrent execution G_1 of Fig. 2. We have four instructions. The requirement of processor 2 is that a write of value 1 to address a exists. Besides that, it imposes no ordering with respect to any other instruction. Same with processor 3. Processor 1, on the other hand, requires that the read of value 1 precede the read of value 2 at address a. This has an indirect effect on the write ordering: w(a,1)[6] must precede w(a,2). Hence, a logical order, in case it exists, must satisfy all these requirements. For this instance, 3, 1, 4, 2 is the required logical order.

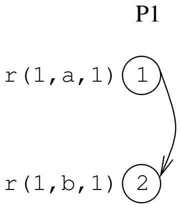

Figure 3. Sample concurrent execution, G_2.

Now, look at a snippet of a concurrent execution G_2, in Fig. 3. One requirement is that r(1,a,1) precede r(1,b,1). It is also required that w(a,1) and w(b,1) exist. However, it does not seem to relate these writes. We might conclude that this is all the requirement enforced by this pair of reads to different addresses and we would be wrong!

The trick is in negation. Instead of expressing the requirements as *enforced* orderings, we could express them

[6] This is a shorthand for w(p,a,1) for some $p \in P$. Since we are dealing with unambiguous runs exclusively, there is at most one such write.

as *forbidden* orderings. For instance, in Fig. 2, we could say that processor 1 forbids the ordering where `w(`a`,2)` precedes `w(`a`,1)`. In the case of binary orderings, the difference is superfluous. However, for Fig. 3, if we say that, for any other write to b, `w(`b,d`)` such that $d \neq 1$, we cannot have `w(`b`,1)` precede `w(`b,d`)` when both precede `w(`a`,1)`, we introduce a new requirement.

It turns out that a formalization of the above ideas to form a set of impossible orderings over the writes of a concurrent execution helps us form a new problem, equivalent to interleaved-sequentiality checking. For a given concurrent execution, we define a set of constraints which basically combines all possible kinds of forbidden orderings for the execution.

Theorem Let G_c be a legal (unambiguous) concurrent execution and $\mathcal{CS}_c$ be its constraint set. Then, G_c is interleaved-sequential if and only if $\mathcal{CS}_c$ is satisfiable.

Previous work on interleaved-sequentiality checking either completely ignored the problem of finding the subset of the execution that violated the property [26], or tried to characterize it in terms of cycles [20]. With the constraint sets, we can define what it means to have a minimal subset of a non interleaved-sequential (i-s, for short) concurrent execution such that the minimal subset still is a violating execution, but any execution subset of it is not.

Let us examine the concurrent execution G_3 that is not i-s, given in Fig. 4[7] Assume that a logical order is being searched for this execution. Starting from the requirement of processor 2, we see that `8 (w(2,`a`,1))` must be ordered before `9 (w(2,`a`,2))` since `(8,9)`$\in E_c$. This ordering implies that `2 (r(1,`a`,1))` is ordered before `9 (w(2,`a`,2))`. Since `(1,2)`$\in E_c$ and `(9,10)`$\in E_c$, we have to order `1` before `10` which implies the ordering of `4` before `10` (hence the dashed line from `4` to `10`). Continuing in this manner, we eventually come to a point where we have to order `5` before `12`, which would violate a property of interleaved-sequentiality. A similar analysis could be performed for the dotted lines which is the result of ordering `12` before `6` due to the edge `(5,6)`$\in E_c$.

Given the above example, it is not clear how, solely based on cycles, we can pick a minimal set of vertices that still is not i-s. Clearly, just picking, say, vertices `4` and `10` because there is a cycle between the two will not be correct. Actually, this concurrent execution is minimally non i-s, that is, any removal of a vertex from the graph would make the remaining subset i-s. This is precisely where we can use the constraint set.

Definition: Let G_c be a non i-s concurrent execution and $\mathcal{CS}_c$ its constraint set. Then a minimal constraint set, subset of $\mathcal{CS}_c$, is a set which itself is unsatisfiable but any proper subset of it is not.

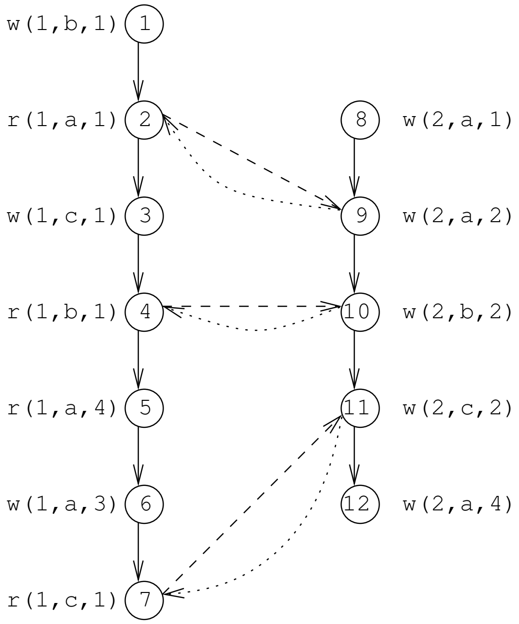

Figure 4. Sample non-i-s concurrent execution G_3 illustrating cycles and the minimal set. The dashed lines are the result of ordering `w(`a`,1)` before `w(`a`,2)`. The dotted lines are the result of ordering `w(`a`,4)` before `w(`a`,3)`.

Note that there can be more than one minimal set for a given G_c. This definition allows us to define minimality with respect to the constraint set.

For P, A, D all finite, we prove in [11] that the size of any minimal instruction set of any non-i-s unambiguous concurrent execution is bounded. An implementation has a non i-s unambiguous concurrent execution if and only if there exists a run that does not visit any state more than $4|A|^2(|D|+1)^3$ times, generating a non i-s concurrent execution. *This bound makes it possible to have a decision procedure for detecting non i-s unambiguous concurrent executions in a memory system.*

Even though, this result might seem intuitively trivial since there are only finitely many different write events in the (infinite) set of unambiguous executions for finite values of P, A and D, it was not possible to obtain it using the previous methods based on cycle analysis. The most important aspect is that we have not resorted to making assumptions about the concurrent executions, about certain relations between instructions and responses. There is also an interesting open problem. When we talk about constraints, we do not take into account the fact that the machine that

[7] We have added some edges - dotted and dashed lines - that are not part of the concurrent execution for illustration purposes. These edges actually would have been added by the algorithm given in [20] or [23].

generates the execution is actually finite-state. Due to this finiteness, the executions cannot be arbitrary but follow a certain regular pattern, which so far we have not been able to characterize. That might render the definition of a certain equivalence relation, having only a finite number of equivalence classes, possible.

6. Conclusion

This paper attempts to allay the notion that the issue of decidability of sequential consistency is a closed chapter. It offers a transducer based definition of sequential consistency that addresses implementation constraints. In this setting, decidability is still an open problem. However, by adopting a constraint-based approach, one can, for unambiguous executions, obtain a decision procedure. The procedure for generating these constraints itself forms an alternative to analyzing cycles (e.g., [27]), and may form the basis for a more efficient SAT-based execution checking approach than reported in [28]. All these will form the subject of our continued research.

References

[1] N.R. Adiga. An overview of the bluegene/l supercomputer. In *Conference on High Performance Networking and Computing: SC2002*, pages 60–60, 2002. (with 30 co-authors).

[2] William Pugh. The java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(1):1–11, 2000.

[3] Shared memory consistency models and protocols, October 2004. Invited Tutorial by Ching-Tsun Chou, Steven German, and Ganesh Gopalakrishnan. `http://www.cs.utah.edu/~ganesh/presentations/fmcad04_tutorial2/`.

[4] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. In *MEMOCODE*, 2003.

[5] David L. Weaver and Tom Germond. *The SPARC Architecture Manual – Version 9*. P T R Prentice-Hall, 1994.

[6] A Formal Specification of Intel(R) Itanium(R) Processor Family Memory Ordering, 2002. `http://www.intel.com/design/itanium/downloads/251429.htm`.

[7] Denis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[8] S. P. Midkiff, J. Lee, and D.A. Padua. A compiler for multiple memory models. *Concurrency, Practice and Experience*, 16(2):197–220, February 2004.

[9] Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. In *Symposium on Logic in Computer Science*, pages 219–228. IEEE, 1996.

[10] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[11] Ali Sezgin. *Formalizing and Verification of Shared Memory*. PhD thesis, The University of Utah, 2004. `http://www.cs.utah.edu/~ganesh/unpublished/sezgin_phd_04.pdf`.

[12] Thomas Henzinger, Shaz Qadeer, and Sriram Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification99*, volume 1633 of *Lecture Notes in Computer Science*, pages 301–315, Trento, Italy, July 1999. Springer-Verlag.

[13] Ratan Nalumasu. *Formal design and verification methods for shared memory systems*. PhD thesis, University of Utah, Salt Lake City, UT, USA, December 1998.

[14] Jesse D. Bingham, Anne Condon, and Alan J. Hu. Toward a decidable notion of sequential consistency. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 304–313. ACM Press, 2003.

[15] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Transactions on Parallel and Distributed Systems*, 14(8), August 2003.

[16] Ali Sezgin and Ganesh Gopalakrishnan. On the definition of sequential consistency. Submitted for publication. Preliminary version at `http://www.cs.utah.edu/~ganesh/unpublished/sc_definition.pdf`, 2004.

[17] M. Frigo. The weakest reasonable memory. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 1998.

[18] Prince Kohli, Gil Neiger, and Mustaque Ahamad. A characterization of scalable shared memories. Technical Report GIT-CC-93/04, College of Computing, Georgia Institute of Technology, January 1993.

[19] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, December 1995.

[20] Ratan Nalumasu. *Design and Verification Methods for Shared Memory Systems*. PhD thesis, Department of Computer Science, University of Utah, 1999.

[21] Anne E. Condon and Alan J. Hu. Automatable verification of sequential consistency. In *13th Symposium on Parallel Algorithms and Architectures*, pages 113–121. ACM, 2001.

[22] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *Proceedings of the 11th International Conference on Computer-aided Verification (CAV)*, number 1633 in Lecture Notes in Computer Science, pages 301–315. Springer-Verlag, July 1999.

[23] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. Technical Report 176, Compaq SRC, December 2001.

[24] Jean Berstel. *Transductions and context-free languages*. Teubner, 1979.

[25] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[26] William W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, Inc., 1992.

[27] W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[28] Ganesh Gopalakrishnan, Yue Yang, and Hemanthkumar Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *CAV (Computer Aided Verification)*, pages 401–413, 2004. LNCS 3113.