FORMALIZATION AND VERIFICATION OF SHARED MEMORY

by

Ali Sezgin

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2004

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Ali Sezgin

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Ganesh G. Gopalakrishnan

Allen E. Emerson

Matthew Flatt

Wilson Hsieh

Ratan Nalumasu

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Ali Sezgin in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ganesh G. Gopalakrishnan
Chair: Supervisory Committee

Approved for the Major Department

Christopher R. Johnson
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Shared memory verification, checking the conformance of an implementation to a shared memory model, is an important, albeit complex on many levels, problem. One of the major reasons for this complexity is the implicit manipulation of semantic constructs to verify a memory model, instead of the desired syntactic methods, as they are amenable to be mechanized. The work presented in this dissertation is mainly aimed at reformulating shared memory verification through a new formalization so that the modified presentation of the problem manifests itself as purely syntactic.

(Shared) memories are viewed as structures that define relations over the set of programs, an ordered set of instructions, and their executions, an ordered set of responses. As such, specifications (basically memory models that describe the set of executions considered correct with respect to a program) and implementations (that describe how an execution relates to a program both temporally and logically) have the same semantic basis. However, whereas a specification itself is described as a relation, an implementation is modelled by a transducer, where the relation it realizes is its language. This conscientious effort to distinguish between specification and implementation is not without merit: a memory model needs to be described and formalized only once, regardless of the implementation whose conformance is to be verified.

Once the framework is constructed, shared memory verification reduces to language inclusion; that is, checking whether the relation realized by the implementation is a subset of the memory model. The observation that a specification can be approximated by an infinite hierarchy of finite-state transducers (implementations), called the memory model machines, results in the aforementioned syntactic formulation of the problem: regular language inclusion between two finite-state automata where one automaton has the same language (relation) as the implementation and the other has the same language as one of the memory model machines.

On a different level but still related to shared memory verification, the problem of checking the interleaved-sequentiality of an execution (an execution is interleaved-sequential if it can be generated by a sequentially consistent memory), is considered. The problem is transformed into an equivalent constraint satisfaction problem. Thanks to this transformation, it is proved that if a memory implementation generates a non interleaved-sequential and unambiguous execution (no two writes in the execution have the same address and data values), then it necessarily generates one such execution of bounded size, the bound being a function of the address and the data spaces of the implementation.

To Şişyanak

CONTENTS

CHAPTERS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This work, which is somewhat unorthodox as it strives to develop a novel theoretical framework in lieu of the existing ones, could not have been possible in an environment lacking patience, freedom and trust. Even though I believe that these should be standard in an academic institution, such is not the case, and I am grateful to my advisor, Dr. Ganesh Gopalakrishnan, who has been instrumental in making me believe that such ideal cocoons for research can and do exist. I have been a gracefully embraced PhD student.

Due to personal reasons, I have had certain timing constraints, or depending on from where you look, problems. I would like to thank my committee members for putting up with all the requests, which at times could have well been perceived as whims.

I have received several important comments on my work. Besides the obvious one, Dr. Gopalakrishnan, Dr. Stephan Merz of LORIA, had the kindness to go through the very rough initial drafts and had considerable impact on the shaping of the formalization. Dr. Wilson Hseih and Dr. Ratan Nalumasu made me realize not every aspect of my research was as important. Prosenjit Chatterjee and Ritwik Bhattacharya, my comrades on this arduous road of graduate study, have been points of both discharge and recharge. I cannot overstate the importance of their role in my work.

Of course, there are always those who have nothing to do with my academic work, but whose absence I cannot bear: my friends and my family. I would like to thank them for reminding me that there is a world out there whose existence I have severely suspected on more than several occasions.

Finally, I am one of those who think without passion nothing is worth doing. Ipek, my wife, my companion, through her own mysterious ways, kept me strong enough to like what I was working on. If it was not for her, I would probably be writing a resignation letter at a company I would have loathed for devouring my life and creativity, instead of this acknowledgment for a work I am proud of and absolutely have no regrets whatsoever!

CHAPTER 1

INTRODUCTION

One of the prominent characteristics of today's society is its infatuation with speed. Fastest cars are pitted against each other as entertainment for the weary minds. We admire, usually to the point of adoring/worshipping, the fast minds of professional athletes; we are fascinated by the way these sports figures minimize the time between perception and action. Arguably, the most popular events in any kind of race are the ones with short duration; we want to see fast human beings and we want the result fast. The examples can be piled up into a huge mountain of evidence to the present obsession for speed.

Being part of, in fact not arbitrarily but essentially, today's society, computers are no different. We want our internet connection to be as fast as possible, we want our computer to crunch numbers at a speed that was inconceivable only a few decades ago. And the majority of research on computer production is aimed at this aspect: faster processors, faster system buses, faster memories, faster communication, etc.

Unfortunately, the speed of a computer that can be summarized as the number of operations it can perform in unit time cannot grow without limit. There are physical boundaries which are insurmountable. The major remedy for this and the hope to satisfy the ever lasting thirst for speed seems to lie in plurality.

It is nothing but normal to expect to have a job done in a shorter period of time when everything relating to the job is kept the same while the work force is increased. Construction sites, military, public transportation are but some instances of this principle. The same method, increasing the computing power through the introduction of more and more computing devices, can be and has been applied to computing.

Just combining a few computers in some ad hoc fashion and hoping them to run in harmony would be naive. For most cases, the processing units should communicate information among them. The physical connection frame, the anticipated characteristics of the work load, the kind of operations to be performed are all aspects that affect the kind of information to be shared. Without getting into much detail, we can say that there are two main paradigms for the dissemination of information in such a system: message-passing architectures and shared memories.

In a system based on message-passing, the processing entities communicate via messages. There are communication protocols to which they are expected to comply. Depending on the connection topology, the messages could be broadcast or be peer to peer. The burden, most of the time, is on the programmer or a system level programmer.

As an alternative favoring the programmers, shared memories have also been proposed. In this case, there is a memory system, either physically or logically shared, and the processing units communicate through this memory system by reading or writing to certain memory locations. The values read/written usually have specific meanings and enable the programmer to arbitrate the operation of the processing units. However, this abstraction comes at a price: it is not always clear what takes place in real time. This has the undesired effect of introducing some sort of nondeterminism to the way the ”shared memory” behaves. There is ”always” a way out, or so has the past made us believe. This time, it comes in the notion of *shared memory model*.

1.1 Shared Memory Models

A shared memory model, simply put, is a restriction on the possible *outputs* that a shared memory might generate for any *input*. A point of contention immediately occurs: what exactly is meant by the input/output of a memory?

The common approach, albeit controversial, is to see the input as a program projected onto its memory access operations. This point is controversial because whether the input to a memory is actually the unfolding of a program or not depends on what is perceived as memory. Memory could be taken as the combination of the operating system, compiler, communication network and a collection of physical storage devices. This would represent the view of a programmer. Or it could be merely a storage device. In the former case, the input and output will indeed represent the unfolding of a program. In the latter, however, the instructions could be reordered. For instance, a compiler might and most likely will rearrange the instructions such that a better utilization of time is achieved. In such a case, we can no longer talk about a program per se.

Regardless of what a memory really represents, one thing remains invariant. There is always a stream of instructions presented to the memory and the memory in return generates a stream of responses. For the sake of simplicity, it is common practice to assume that there are two types of instructions: read instructions, instructions that query the contents of a location, and write instructions, instructions that update the contents of a location.

The input to a memory, then, becomes this stream of instructions. In the case of a single user of a memory, the input is simply represented as a string over a suitable alphabet. In the case of multiple users, which is the case for shared memories, the input becomes a collection of strings; one string per user.[1]

As for the output, the memory is expected to generate suitable responses for the instructions it accepts. For instance, for a read instruction, the memory should return the value of the location queried by the read instruction. For a write instruction, the response is not as obvious. It will be assumed, without loss of generality,[2] that the memory generates an acknowledgment, which merely will mean that the memory has input the instruction.

[1] As we will see in the next chapter, this is the most abstract view of a memory; usually, more detailed representations are employed.

[2] For the case where write instructions do not generate any response, see the explanation in [37].

Much like the input, the output is also a stream of responses. In the single user case, we will have a single string, whereas for shared memories, the output will be a collection of strings, one for each user.[3]

From now on, programs (executions) will be understood as inputs (outputs) to a (shared) memory as explained above.

When there is only one user, the expected behavior is not complicated: a query of a location should return the most recent value written into that location. The notion of *most recent* should be clear: the temporal ordering of queries and updates as they are input into the memory. This is always implicitly assumed; we can say that there is a single memory model for the single user case!

Things do get complicated, however, when we think of multiple users of a memory, which is precisely the root of confusion for shared memories. Even if a global temporal ordering can be defined and there are many systems where such an ordering is impossible to define, almost all the speed-up, hence the raison d'être of shared memories, would be limited by the memory itself!

Let us go back to the first sentence of this section. It should now be clear what we mean by a shared memory model: for any program, it defines a set of allowed executions. A shared memory model, therefore, restricts the level of non-determinism for shared memories. The formal definition will have to be deferred until the next chapter.

Shared memory models are abstract entities; they are not expected to be fully implemented. The idea is to design a shared memory system that operates under a certain shared memory model; for any program, what the system generates should be among the allowed executions defined by the model, but it is not expected to generate all possible executions. This point, as we shall see in the next chapter, makes us distinguish the notion of a model and that of a system.

When a memory system is designed, the designer either has a specific model to which the system is to conform, or defines a new model and claims that the system follows the model. In both cases, unless we assume the infallibility of the designer,

[3] See Chapter 2.

which we never should, a new problem asserts itself: how can a memory system be shown to comply with a memory model? The answer to this question takes us to the next section.

1.2 Formal Verification and Shared Memory Models

The objective of any kind of verification is simple: to obtain a *convincing* argument that a certain property holds for a certain structure. The convincing argument could be done by presenting evidence, by testimony, by comparison, by investigation, etc. In what is called the real world, these discussions almost never form an argument that convinces all. Some of the reasons are lack of ground rules, not being able to reach a consensus on basic assumptions, terms not having constant meaning but being contextual. As a remedy, since the Ancient Greeks, an alternative domain has been formed: mathematics/logic.[4] Once the assumptions, or the axioms, and the deduction rules are set, a proper argument, or a proof, transcends subjectivity and becomes a demonstrable truth.

Formal verification forms a bridge between the real world and this ideal realm. It is concerned with real world objects, such as microprocessors, communication protocols, software code, and with real world properties, such as the liveness of a system, deadlock freeness of a protocol. The argument, in return, is carried on in the mathematical domain; so these structures and properties are represented as mathematical objects. Furthermore, the steps in the argument depend solely on its form, or syntax, and as such, become amenable for mechanization. Therefore, there are two important relations: the first one relates *real* entities to mathematical objects; the second one relates these objects to syntactical structures. Usually, however, these two relations are coalesced into a single relation where a structure is defined syntactically and its semantics is provided by a well-known mathematical theory. This step is known as the formalization of a problem.

[4] Whether logic is more general than mathematics or the opposite, has been and still is the source of much controversy. Here, we tend to take the two together.

Since we are dealing with shared memories, next we will see the previous formalization efforts for shared memories.

1.3 Previous Work on Formalizations

The work on shared memories might not be as popular as, say, graph theory, but it is not exactly a rarely visited topic either. It would just not make much sense to list all the relevant work on this topic, one after the other, in no coherent order. Instead, we will try to categorize previous work according to the level of formalization which depends on what ultimate motivation it has. Even though, the bounds are more often than not quite blurred, it can be argued that there are three major camps: the semiformalization, used mostly by people concerned with the design/implementation of memory systems, the specification-oriented formalization, used by people interested in comparing different shared memory models with each other, and finally, verification-oriented formalization, used by people who try to come up with efficient and mechanizable methods for the verification of a specific or arbitrary shared memory models for arbitrary shared memory systems.

1.3.1 Semiformalization

Included under this rubric are the works that primarily focus on designing new shared memory systems. The common characteristic of this type of work is its dependence on meta-narrative to explain how a memory works. Consider the following excerpt from one of the better-known papers [4] in this camp:

> A write is said to be globally performed when its modification has been propagated to all processors so that future reads cannot return old values that existed before the write. A read is globally performed when the value to be returned is bound, and the write that wrote this value is globally performed.

It might very well be the case that this sentence poses no problem for a designer, but we think that a definition of this kind cannot be deemed *formal*. Expressions like “existed before” or “bound” are semantic in nature. It is assumed that the system is modelled by a state machine and at any state, these properties have truth values. However, the truth values are assigned not based on a syntactic definition but most likely, by the designer himself/herself. This has the risk of making any kind of verification on such a system dubious; the verification is as correct as the designer is correct at assigning those truth values. Instead, we ultimately want a complete syntactic model which should not be annotated semantically.

Implied by this kind of definition, are formalizations that explain the operation of a shared memory system based on temporal orderings of operations. Typically, a shared memory model is given. A set of sufficient conditions for any system satisfying this memory model is devised. These conditions dictate which instruction can be issued or which operation can be completed. Consider the following quote[27]:

> In a multiprocessor system, storage accesses are *strongly ordered* if
> 1. accesses to global data by any one processor are initiated, issued and performed in program order, and if
> 2. at the time when a STORE on global data by processor I is observed by processor K, all accesses to global data performed with respect to I before the issuing of the STORE must be performed with respect to K.

Naturally, any formalization used in these approaches will have a time information, be it relative or absolute, and semantically annotated events, as explained above, will have to be ordered. Some examples include [2, 5, 6, 7, 26, 29, 45, 56, 58, 61, 62].

It is worth noting that strong ordering of the above quote was designed as a sufficient condition for sequential consistency. It turned out to define a memory system not comparable to sequential consistency![3]

1.3.2 Specification Oriented Formalization

Formal specification, as an active research area, seeks to remove from systems ambiguities which do cause misunderstandings or contradictions when their descriptions are given in an informal manner, that is, using natural language descriptions. In this sense, formalization itself becomes the end result.

In shared memories, specification has been used mainly to provide a taxonomy of shared memory models whose semantic differences or similarities are better captured in a unified formalization. The work done in this vein can be further divided into two: memory as a transducer, memory as a generator.

The first class models memory as a system that is characterized by its input and its output. Each operation, read or write, is seen as a process whose start and end points correspond to the invocation of the operation and the termination thereof, respectively. Some examples include [14, 8, 13, 30, 31, 36, 37].[5]

The second class, on the other hand, sees and characterizes memory by its set of executions. It is not hard to see that, if a many to one corresponding exists between the set of responses a memory generates (memory's output set) and the set of instructions it receives (memory's input set), the set of responses generated for a particular execution plus the information on *program ordering*[6] can be used to extract the *program*.[7] Based on this observation, some work have opted to use only executions as the basis for the formalization of memory models. In this execution-based approach, a memory model is usually described through what it shall not generate (or accept). Examples include [9, 10, 28, 24, 48, 49, 41, 54, 60].

1.3.3 Verification Oriented Formalization

Finally, there is the kind of work that this dissertation belongs to. The works under this category try to develop a general enough approach to the formal verification of shared memory models. Unlike the first approach, implementation details are usually abstracted. Unlike the second, the execution itself is viewed as a process. This process is also modelled using mathematical structures. More often than not, a finite state automaton is used. That in turn implies that a memory is usually seen as a set of strings, or traces in certain contexts. This is akin to the execution-based approach of the previous class; the notable distinction being the introduction of the generator itself as a part of the problematic. Some examples are [12, 16, 18, 17, 20, 21, 23, 25, 34, 35, 39, 40, 42, 46, 47, 51, 52, 53, 59].

[5] Some of these, namely [8, 31], actually contain some verification results but they do not try to generalize the results for arbitrary systems and/or models. Hence, they are not considered to be verification-oriented as they do not try to obtain a methodology.

[6] Program order, a rather habitual naming, is the order of issuing instructions per processor.

[7] Of course, some untold assumptions must hold, such as, the memory does not generate responses arbitrarily and only to the instructions it has accepted.

Unfortunately, a verification methodology derived from an execution-based approach has some important inadequacies. As we will argue in the next section, abstracting away the input part might lead to results not corresponding to the original problem.

1.4 Presented Work

The title of the dissertation explicitly states that we will be concerned exclusively with the problem of formal verification in the context of shared memories. In the light of the previous argument, this means that we have to have a formalization and a certain methodology for the problem of shared memory verification. The first part of the dissertation, composed of the following two chapters, is indeed following this pattern. The somewhat odd chapter out, Chapter 4, is a more detailed look into a specific shared memory model. Let us briefly summarize these.

1.4.1 New Formalization

As we have already demonstrated, the area of shared memory formalization does not really lack formalization. There seems to be many different approaches; one must surely be able to pick the suitable formalization and use it for whatever one sees fit. The need for a new formalization actually originated from a previous formalization used for a certain problem.

In their well-known paper [12], Alur et al. obtain some very strong results about the verification of certain shared memory models.[8] The result with arguably the most important repercussion is what has come to be known as the undecidability of sequential consistency. In this paper, it has been proved that the class of sequentially consistent languages is not regular. This result has since been used as the evidence to the impossibility of developing an algorithm which can decide whether a given finite state shared memory system is sequentially consistent or not. Almost all of the work done after [12] cites this work and tries to define a maximally decidable class of finite state shared memory systems. For instance,

[8] Strictly speaking, according to our formalization introduced in the next chapter, linearizability is not a shared memory model.

in [35], it is claimed that "[even] for [finite state systems], checking if a memory system is sequentially consistent is undecidable." We have, however, suspected the application of [12]. The link from this undecidability result to the perceived undecidability of shared memory verification is fallacious, fallacy being a direct result of the execution-based formalization used. There are two related issues. One is to do with how a memory model is defined; the other is to do with the absence of program, or input, in the formalization.

A shared memory is to generate an execution for any syntactically correct program. A shared memory that generates an execution for only a proper subset of programs would be violating any kind of *sensible* correctness criterion. But the argument used for the result of [12] does precisely that. As long as the execution-based approach where a memory is viewed as a generator is used, the undecidability result for shared memory verification follows. As an analogy, it would be like claiming the nonregularity of $\Sigma^\star$ because its subset Σ^p for p prime is not regular. Actually, Alur et al. seem to be aware of this fact when they say in [12]:

> ... 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*.

We have pointed out the abstracting away of program from the formalization as the second reason as well. This is due to the fact that without the notion of a program, or input, certain characteristics of finiteness of the shared memory system cannot be expressed.

Consider the following regular expression[9]:

$$\texttt{w(1,1,2)}\ \texttt{r(1,1,1)}^*\ \texttt{r(2,1,2)}^*\ \texttt{w(2,1,1)}$$

[9] w(p,a,d) (r(p,a,d)) denotes the writing (reading) of value d to (at) location a by processor p.

As long as the definition of [12] is concerned, a shared memory system with the above regular expression is sequentially consistent.[10] Furthermore, since this is a regular expression, it is claimed that it is the output of a finite state shared memory system. However, a finite state and sequentially consistent system cannot generate all the strings that belong to this regular expression (see Appendix).

Based on this, we clearly want to have a formalization that also represents the transduction nature of memory; we should have both the program and the execution. Furthermore, the second reason above implies that the temporal ordering of input and output is relevant and should not be abstracted away.

In the light of all these, we have not so much developed a novel formalization as picking suitable parts from each formalization presented above. We will model a shared memory model as a certain relation over programs and executions. This relation will be called *specification* and the emphasis will be on what it contains and not how that relation can be realized. A shared memory system, in turn, will be based on a specific mathematical structure, a transducer, which might be considered as a variation of the basic concept of automaton. A transducer satisfying certain properties will be the mathematical equivalent of a shared memory system, and will be called an *implementation*.

Finally, we should note that a few formalizations [8, 38] are very close to our formalization. The major difference is that in those works, memory system is assumed to complete its instructions in order and pipelining of instructions is not allowed. Specifically, in [8], this results in a restricted definition of sequential consistency, which is not equivalent to the original definition given in [43]. The assumption that the processor does not submit an instruction until it receives the response for the previous instruction, as in [38], removes a major difficulty in the formalization. However, that assumption no longer reflects the real world systems.

[10] At this point, we do not want to get into the specifics of sequential consistency. The reader can review the definition of sequential consistency given in the next chapter and see for himself/herself that this regular expression indeed forms a set of strings each of which belong to a sequentially consistent specification.

The next chapter will give an idea to the reader about the complexity of formalizing without this assumption.

1.4.2 Verification as Language Inclusion

Once the formalization is done, we will demonstrate how we can make use of the new approach. The objective, since the beginning of this research, has been the development of a framework where the formal verification of shared memory models could be automated. Our emphasis on a language based formalism is primarily due to this desire. In Chapter 3, we will indeed formulate the problem as a language inclusion problem: a shared memory system satisfies a certain shared memory model if its *language* is contained within the language of a machine, element of a certain class of machines defined according to the shared memory model itself. We will demonstrate this method using lazy caching [8] as the memory system and sequential consistency [43] as the memory model.

We should tell that the method as of now is not complete. We were not able to develop a method which would verify a memory model for a memory system if and only if that memory system conforms to that memory model. There might be instances where the system conforms to the model, yet the language inclusion fails to hold. But unlike previous work on this area, we claim that the problem is open and not undecidable as has been the general perception.

1.4.3 Debugging Sequential Consistency

Sequential consistency is not an arbitrary shared memory model. It has been the first to be, albeit informally, proposed as a correctness criterion for shared memory systems. It is not abnormal, then, for us to concentrate on this memory model. Unlike Chapters 2 and 3 whose results are not confined to sequential consistency but hold for all memory models and systems, Chapter 4 deals exclusively with sequential consistency.

Formal verification as language inclusion can be seen as the *sufficiency* approach. When a system satisfies the inclusion, it is proved that the system satisfies its memory model. However, the failing of the language inclusion is inconclusive; no result can be drawn without additional and possibly different work.

We can also approach from the other end. We can generate a set of tests which would try to find violating executions of the memory system. We call this the *debugging* approach.

In Chapter 4, we obtain a strong result for the debugging approach. We are able to prove that for a given finite state shared memory system, it is decidable to check whether it has an unambiguous[11] execution that violates sequential consistency. This result is obtained through a transformation of the original problem to a constraint satisfaction problem. We hope that this transformation also sheds some light on to the intricacies of sequential consistency.

[11] An execution in which there do not exist two different write operations with the same location and data values.

CHAPTER 2

FORMALIZATION OF SHARED MEMORIES

In this chapter, we will develop a new formalization for shared memories. This formalization is based on the theory of (rational) transduction, a topic in formal language theory (for introductory texts, see, for instance, [15, 55]). In this formalization, we will distinguish *specifications* as shared memory models (the definition of which program/execution pairs are allowed) from *implementations* as the descriptions of how shared memory systems behave. The latter is modelled as a (length-preserving) rational transducer, whereas, for the former, we do not require any particular approach.

We show that as long as the "user" and the "memory" are finite entities, we can do away with *arbitrary* implementations and work on a canonical model instead. As we shall see, the biggest challenge in moving from specifications to implementations is the formulation of the mapping between input (instructions) and output (responses) symbols using only a finite set of *tags*, or colors as we will call them in this work.

2.1 Introduction

A formalization of a real world entity entails an inevitable abstraction. Inevitability is due to the (perhaps debatable on a philosophical level) infiniteness of the real world and the preferred finiteness of the target domain and the finiteness of the abstraction process itself which has to terminate in finite time. The crucial decision in formalization, therefore, is to choose what to abstract and what to represent. For instance, names are almost never represented. In formalizing a transistor, we do not care whether a particular transistor in a particular design is called $T301$ or *Faust*; they are modelled by the same structure as long as they are deemed identical on their operational specifications. For some aspects, the decisions are rather trivial. Then again, having the mathematical structure represent certain information about the real object or not can make all the difference. We have seen in the first chapter that the absence of program information in the formalization might and will cause one to reach inaccurate conclusions.

Another notable aspect about the formalization of shared memories is one of our own making. We find it appropriate to represent shared memory models and shared memory systems on two different levels of abstraction. We view a shared memory model as a relation. How that relation is to be realized should not be part of the definition of the shared memory model. Hence, a shared memory model should be a nonoperational structure. That an equivalent operational structure can be constructed is irrelevant to the formalization of the memory model. A shared memory system, on the other hand, should fore and most describe how the system behaves; hence the need for an operational structure. Of course, these two different levels of abstractions should be related to each other.

With these points in mind, we argue that there are four levels of abstraction for shared memories. We will now briefly discuss these levels.

1. **Abstraction Level One**: This corresponds to the highest level where we represent the program and its execution as two isomorphic partially ordered sets. A typical representation is given in Fig. 2.1.

   Specifically, we do not have any information on the temporal ordering of instructions or responses besides that of the ordering of instructions issued by

```
   P1       P2                P1        P2

 w(1,1)    r(1)             w(1,1)    r(1,0)

           r(1)                       r(1,1)
```

Figure 2.1. A program and its execution represented in the most abstract level.

the same processor. That is, in Fig. 2.1, we know that the first read instruction of processor 2 is indeed issued before the second read instruction but we do not have any information about their respective ordering of completion or how they are ordered with respect to other instructions or responses of other processors.

For the following levels, let i_1 denote the write instruction of processor 1, i_2 and i_3 denote the first and second read instruction of processor 2, respectively. Let r_1, r_2 and r_3 denote the responses corresponding to these instructions.

2. **Abstraction Level Two**: The next level adds some more information about temporal ordering. A possible representation of Fig. 2.1 is given below:

$$i_2\ i_1\ i_3,\ r_1\ r_3\ r_2,\ \texttt{3 1 2}$$

In this representation, there are three strings. The first one represents the program. Instead of giving only per processor issuing order, this string also totally orders instructions issued by different processes. It is assumed that a symbol precedes (or is in the prefix of the subword up to) another symbol if and only if the former is issued before the latter.

The second string represents the execution. A similar total order is given for the responses as well. This time the order is done according to their time of completion.

In the previous abstraction level, since the isomorphism between the program and the execution was clear from the formalization, we did not need additional structures to represent which response was to which instruction. However, this is not the case for this level. The last string, a string of numbers, which as we will see, represents a mapping between the string of instructions and the string of responses, takes care of this isomorphism. Its semantics will be given later in this chapter, but for now, it suffices to point out that the mapping in this level maps the first instruction to the third response, the second to the first and the last to the second.

3. **Abstraction Level Three**: The third level is less abstract from the previous level not so much because of the amount of information it represents as the way the same information is represented. We still have the same information about instructions and responses, and the temporal ordering of issuing and completion. But, instead of using the infinite set of natural numbers to represent the mapping between instructions and responses, we use a finite set of symbols. Below, we give a possible representation of Fig. 2.1:

   $$(i_2, c_1)(i_1, c_2)(i_3, c_3),\ (r_1, d_1)(r_3, d_2)(r_2, d_3)$$

   $$\varphi_c((c_1, d_1)\ (c_2, d_2)\ (c_3, d_3)) = 3\ 1\ 2$$

   Note that, this time we do not have the third component. Instead, we have tagged each instruction and response with the elements of a finite set; c_i and d_i, not necessarily different, all belong to the same (finite) set. Additionally, we now have included a function, φ_c, which maps a string of pairs over these elements to a string of natural numbers, which in turn, is nothing but the mapping of the previous abstraction level.

   The motivation behind this, as will be discussed later, is the appeal to the finiteness of a shared memory system, or an implementation.

4. **Abstraction Level Four**: This level is the lowest level and has the most information about the program and its execution. The additional information, compared to the previous level, is the temporal ordering of instructions and responses. With this formalization level, the relative temporal ordering of an instruction or a response is completely known. A possible representation of Fig. 2.1 is given below:

   $$(i_2, c_1)(i_1, c_2)(r_1, d_1)(i_3, c_3)(r_3, d_2)(r_2, d_3)$$

   $$\varphi_c((c_1, d_1)\ (c_2, d_2)\ (c_3, d_3)) = 3\ 1\ 2$$

   So, for instance, we know that the first response to be completed belongs to the second instruction issued (first instruction of the first processor) and this happens before the third instruction, the second read instruction of the second processor, is issued. Much like the previous abstraction level, we again have the tagging of instructions and responses to determine the mapping between them.

As we will see in this chapter, we choose level two for the specification, level four for the implementation. The latter choice is obvious, as this fourth level actually represents the operation trace (history) or the computation of a shared memory system. The former, however, seems debatable. We could have as well chosen the first level which seems to be a better fit to what we have been explaining about shared memory models. Indeed, the results of this dissertation would be left unchanged, were we to switch to this first level. The choice was made due to the simplicity we get when we want to define the semantics of a trace that belongs to the implementation. We have chosen the third level as the operational semantics of the implementation and mapping that to the second level is trivial. It would have been more cumbersome to use the first level as the semantic basis. Although not really an essential point, the second level had the additional property of being able to distinguish a serial memory from sequential consistency. We prefer to point out this difference even though from a mathematical stand point, there should be none.

In the following section, we will explain the notation used throughout this dissertation, safe for some parts of the fourth chapter. In Section 2.3, we briefly describe rational relations and transducers and provide the theorems that we will make use of. This section is provided mostly to make the dissertation self-sufficient; for more detail on the topic of rational languages and transducers, the reader is referred to [15]. In Section 2.4, which is the main contribution of this chapter, we will develop the proposed formalization. We will start with specifications, explain the intuition for both specification and implementation and develop the generality results for implementations. Sections 2.5 and 2.6 will illustrate the use of the formalization. In the former, we will define sequential consistency. In the latter, we will describe how to model finite instances of the lazy caching protocol as implementations. We end the chapter with a summary of the results.

2.2 Notation

Let $\mathbb{N}$ denote the set of natural numbers. We shall denote the subset $\{1, 2, \cdots, k\}$ of $\mathbb{N}$ with $[k]$. A *permutation* is a bijection from a subset R of $\mathbb{N}$ onto itself. The set of all permutations over R will be denoted by $\mathbf{Perm}_R$. In particular, with an abuse of notation for the sake of simplicity, the set of all permutations over $[k]$ will be denoted by $\mathbf{Perm}_k$. $\mathbf{Perm}$ denotes the infinite union $\bigcup_{k>0} \mathbf{Perm}_k$. A permutation $\eta \in \mathbf{Perm}$ is *bounded by* b if for all $i \in dom(\eta)$, we have $i \leq b + \eta(i)$. A set of permutations is *bounded* if there exists b such that all the permutations in the set are bounded by b. For any $\mathbf{Perm}_R$, the identity function is called the *identity permutation.*

Let an *alphabet*, Σ, be a nonempty set. Its elements are called *letters* or *symbols.* A *string* over Σ is a finite sequence of symbols of Σ. The string with 0 symbols is called the *empty* string, denoted by ε. Let Σ^* be the set of all strings over Σ. In an algebraic setting, as in the next section, Σ^* is also called the free monoid of strings over Σ with respect to concatenation as the associative binary operation and the empty string as the identity element.

For a string σ over Σ and $X \subseteq \Sigma$, let $\sigma = y_1 x_1 y_2 x_2 \ldots y_n x_n$ be a representation of σ such that y_i are strings over $\Sigma \setminus X$ and x_i are strings over X. Then, the projection of σ into X, $\sigma \upharpoonright X$, is the string $x_1 x_2 \ldots x_n$. When X is a singleton $\{x\}$, we will abuse the notation and write $\sigma \upharpoonright x$.

In the case where the alphabet is taken to be $\mathbb{N}$, a string $\mathbf{n} = n_1 n_2 \ldots n_k$ of length k in $\mathbb{N}^*$ will be identified with a mapping $\mathbf{n} : [k] \longrightarrow \mathbb{N}$ such that $\mathbf{n}(i) = n_i$. Usually, $\mathbf{n}$ will be referred to as a sequence rather than a string.

Given a permutation η over $[k]$, consider the sequence $\mathbf{n}$ of length k with $n_i = \eta(i)$ for all $i \in [k]$. Then, $\mathbf{n}$ is called the *canonical representation* of η, where we write $\eta \sim \mathbf{n}$. So, the set of sequences whose mappings are bijections over $[k]$ is isomorphic to $\mathbf{Perm}_k$. Hence, we will use such sequences and permutations interchangeably. For instance, we might talk about a sequence over $\mathbb{N}$ being in $\mathbf{Perm}$.

For any relation R on $D_1 \times D_2 \cdots \times D_n$ and an element $a \in D_1 \times D_2 \cdots \times D_i$ with $i \leq n$, $R(a)$ denotes the set $\{b \in (D_{i+1} \cdots \times D_n) | (a, b) \in R\}$. For a tuple a in $D_1 \times D_2 \cdots \times D_n$, let $\sharp_i(a)$ denote the i^{th} component of a.

For any function $f : A \to B$, $dom(f)$ denotes the domain of f; that is, the subset of A on which f is defined. The image of f, $img(f)$, is the set $\{b | \exists a \in A, f(a) = b\}$. With an abuse of notation, we will also use *dom* and *img* for relations.

When referring to the components of a structure, we will use the name of structure as superscript to address each component. In the case of nested structures, for simplicity, we shall use only the outermost structure's name as superscript when no confusion is likely to arise.

2.3 Rational Expressions, Transductions

Most of the definitions and theorems of this section, more or less standard in the formal language community, are taken from [15].

A *rational subset*, also called a rational language, of Σ^* is either empty or can be expressed, starting with singletons, by a finite number of unions, products, and the plus or star operations. Such an expression is called a rational expression. Kleene's theorem states that, for languages over finite alphabets, rationality and recognizability[1] coincide.

Definition 2.1 *Let X and Y be alphabets. A rational relation over X and Y is a rational subset of the monoid $X^* \times Y^*$.*

A transduction τ from X^* into Y^* is a function from X^* into the powerset of Y^*, written $\tau : X^* \to \mathfrak{P}(Y^*)$. The graph of τ is the relation R_τ defined by

$$R_\tau = \{(f, g) \in X^* \times Y^* | g \in \tau(f)\}$$

Conversely, for any relation[2] $R \subset X^* \times Y^*$, the transduction $\tau_R : X^* \to \mathfrak{P}(Y^*)$ defined by R is given by $\tau_R(f) = \{g \in Y^* | (f, g) \in R\}$.

[1] That a finite state automaton accepts/generates the language.

[2] By a "relation," unless stated otherwise, we will always mean a relation over (finite) strings.

Definition 2.2 *A transduction $\tau : X^* \to \mathfrak{P}(Y^*)$ is rational iff its graph R_τ is a rational relation over X and Y.*

Definition 2.3 [3] *A transducer $\mathcal{T} = \langle I, O, Q, q_0, F, E\rangle$ is composed of an input alphabet I, an output alphabet O, a finite set of states Q, an initial state q_0, a set of accepting states F, and a finite set of transitions or edges E satisfying $E \subset Q \times (I \cup O \cup \{\varepsilon\}) \times Q$.*

For a transition $(s, a, t) \in E$, s, a and t are the *source* state, the *label* and the *target* state of the transition, respectively.

A *run*, $\mathbf{r}$, of $\mathcal{T}$ is an alternating sequence of states and labels, $q_0a_1q_1 \cdots a_nq_n$, such that, the first state q_0 is the initial state $q_0^{\mathcal{T}}$, and for all $1 \leq i \leq n$, we have $(q_{i-1}, a_i, q_i) \in E$. For such a run, we call the sequence $q_0q_1 \cdots q_n$, the *path* of the run denoted by $\mathbf{r}^p$; $a_1a_2 \cdots a_n$, the *label* of the run, $\mathbf{r}^l$; the subword of the label where all and only the input letters are kept, the *input label* of the run, $\mathbf{r}^i$[4]; mutatis mutandis, for *output label*, $\mathbf{r}^o$[5]. We will call the pair $(\mathbf{r}^i, \mathbf{r}^o)$ a label (of the run $\mathbf{r}$) as well. The transducer $\mathcal{T}$ *accepts* a run $\mathbf{r}$, if the final state q_n is in $F^{\mathcal{T}}$. The language of a transducer $\mathcal{T}$ or the transduction realized by $\mathcal{T}$, denoted by $\tau^{\mathcal{T}}$ is the set $\{(\mathbf{r}^i, \mathbf{r}^o) \mid \mathbf{r} \text{ is an accepting run}\}$.

Theorem 2.1 (Thm. 6.1 [15]) *A transduction $\tau : X^* \to \mathfrak{P}(Y^*)$ is rational iff τ is realized by a [finite] transducer.*

A binary relation R over strings is length-preserving if $(\mathbf{f}, \mathbf{g}) \in R$ implies that the lengths of $\mathbf{f}$ and $\mathbf{g}$ are equal. Using these definitions, the following theorem can now be stated.

Theorem 2.2 *A length preserving rational relation over $X^* \times Y^*$ is a rational subset of $(X \times Y)^*$.*

[3] This is a somewhat restricted definition but suits better for this work.

[4] $\mathbf{r}^i = \mathbf{r}^l \restriction I$.

[5] $\mathbf{r}^o = \mathbf{r}^l \restriction O$.

Corollary 2.1 *Given a length preserving rational relation R over $X^* \times Y^*$, there is a finite state automaton with alphabet $(X \times Y)$ that recognizes R.*

2.4 Memory Protocols

Before getting into the specifics of the formalization proposed in this work, we would like to explain our view of (shared) memories. The intuitive picture is summarized in Fig. 2.2. There are two parties involved in the system. One is the *user*, the other is the *memory*. The user could be the collection of processors or threads. It issues memory access *instructions*, such as reads and writes, to be processed by the memory. The memory services the instructions and generates suitable *responses* as output. The *interface* is basically a set of syntactic definitions of instructions and responses that the user and the memory are allowed to use/generate. The interface also defines a set of possible responses for each valid instruction, that is, it makes explicit what it means to generate a *suitable response* for an instruction.

A memory specification defines the behavior of a memory for a given interface. Simply put, the specification relates the input of the memory to its output. From the user perspective, a memory specification is a description of the possible response streams for a given instruction stream.

In the following subsection, we will formalize these ideas and define the interface and the memory formally. For the specification part, we are not concerned about the specifics of the user.

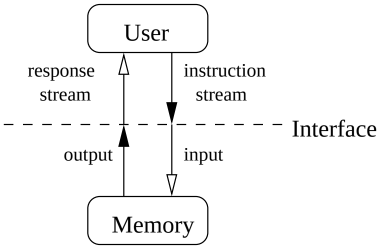


Figure 2.2. User and memory communicate over an interface.

2.4.1 Specification

Definition 2.4 *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.*

The following definition will be useful later for defining parameterized shared memories.

Definition 2.5 *A restriction of a memory interface $\mathfrak{F}$ with respect to a set $\Sigma' \subseteq \Sigma^{\mathfrak{F}}$ is the memory interface $\mathfrak{F}[\Sigma']$ with*

1. *$\mathcal{I}^{\mathfrak{F}[\Sigma']} = \mathcal{I}^{\mathfrak{F}} \cap \Sigma'$.*

2. *$\mathcal{O}^{\mathfrak{F}[\Sigma']} = \mathcal{O}^{\mathfrak{F}} \cap \Sigma'$.*

3. *$\rho^{\mathfrak{F}[\Sigma']} = \rho^{\mathfrak{F}} \cap (\mathcal{O}^{\mathfrak{F}[\Sigma']} \times \mathcal{I}^{\mathfrak{F}[\Sigma']})$.*

It is called lossless in $\mathfrak{F}$ iff $\mathcal{O}^{\mathfrak{F}[\Sigma']} = \{o | i \in \Sigma' \wedge \rho^{\mathfrak{F}}(o, i)\}$.

So, lossless means that for any instruction that is retained in the restriction, all possible outputs defined by the initial interface, can still be generated.

We will actually use a specific memory interface, namely the interface for multiprocessor shared memories restricted to read/write, or rw-interface for short, which is defined as follows:

Definition 2.6 *The rw-interface is the memory interface $\mathcal{RW}$ with*

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

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.*

The rw-interface has only two types of instruction and response. The type R stands for read instructions/responses, and the other type, W, for write instructions/responses. Each write instruction has a unique response. Each read instruction can generate exactly one response from a set. The collection of these sets forms a partition of all possible responses for read instructions. A response is associated with exactly one instruction.

We are now ready to define a *memory specification.*

Definition 2.7 *A memory specification, $\mathfrak{S}$, for a memory interface $\mathfrak{F}$ is the tuple $\langle \mathfrak{F}, \lambda \rangle$, where $\lambda \subseteq ((\mathcal{I}^{\mathfrak{F}})^* \times (\mathcal{O}^{\mathfrak{F}})^*) \times$ **Perm**, is the input-output relation.*

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 $\mathcal{I}$ that might be called a *program* or an *instruction stream*, and the output it generates for this input, a (finite) string over $\mathcal{O}$ that might be called an *execution* or a *response stream.*[6] For each such program/execution pair of the

[6] 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

memory, λ also defines, through permutation, the mapping between an individual instruction of the program and its corresponding output symbol in the execution.[7]

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 2.8 *A shared memory $\mathcal{S}$ is a memory specification for $\mathcal{RW}$.*

Let us define a few exemplary shared memories.

Example 1 *We define the following shared memories:*

- $\mathcal{S}^{\emptyset} = \langle \mathcal{RW}, \lambda^{\emptyset} \rangle$*, where* $\sigma = ((\mathbf{p}, \mathbf{q}), \mathbf{n}) \in \lambda^{\emptyset}$ *implies* $\mathbf{p} \in (\mathcal{I}^{\mathcal{RW}})^*$ *and* $\mathbf{q} = \mathbf{n} = \varepsilon$*.*
- $\mathcal{S}^{U} = \langle \mathcal{RW}, \lambda^{U} \rangle$*, where* $\sigma = ((\mathbf{p}, \mathbf{q}), \mathbf{n}) \in \lambda^{U}$ *implies* $|\mathbf{p}| = |\mathbf{q}| = |\mathbf{n}|$*.*
- $\mathcal{S}^{ND} = \langle \mathcal{RW}, \lambda^{ND} \rangle$*, 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* $j \in [|\mathbf{p}|]$*,* $\eta \sim \mathbf{n}$*.*

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.

[7] By itself, ρ may not be enough to define this mapping, as there might be an input symbol with multiple occurrence in the program, having multiple output symbols that are related to the same input symbol by ρ.

- $\mathcal{S}^{NC} = \langle \mathcal{RW}, \lambda^{NC} \rangle$, *where* $\sigma = ((\mathbf{p}, \mathbf{q}), \mathbf{n}) \in \lambda^{NC}$ *implies*

$$\mathbf{p} \in \{\texttt{(w}_\texttt{i}\texttt{,1,1,1)}\} \cdot \{\texttt{(r}_\texttt{i}\texttt{,1,1)},\texttt{(w}_\texttt{i}\texttt{,1,1,1)}\}^*$$
$$\mathbf{q} \in \{\texttt{(w}_\texttt{o}\texttt{,1,1,1)}\} \cdot \{\texttt{(r}_\texttt{o}\texttt{,1,1,1)},\texttt{(w}_\texttt{o}\texttt{,1,1,1)}\}^*$$

  $|\mathbf{p}| = |\mathbf{q}|$ *and* $\eta(i) = j$ *implies* $\rho^{\mathcal{RW}}(q_j, p_i)$ *for* $\eta \sim \mathbf{n}$.

So far, we have not placed any restrictions on a specification, more specifically on $\mu^{\mathfrak{S}}$. This relation might include $(\varepsilon, \mathbf{q})$ or $(\mathbf{p}, \varepsilon)$ (executions without programs or vice versa), or $(\mathbf{p}, \mathbf{q})$ for $|\mathbf{p}| \neq |\mathbf{q}|$ (the number of instructions and responses do not match). Or we could have $((\mathbf{p}, \mathbf{q}), \eta)$ with equal length $\mathbf{p}$ and $\mathbf{q}$ where η is completely arbitrary, not respecting the response relation defined by the interface. For instance, consider the shared memories defined above. $\mathcal{S}^{\emptyset}$ defines a memory that accepts any program as input but fails to generate any response whatsoever. $\mathcal{S}^{NC}$ accepts only certain input streams; for instance, any program that starts with an instruction other than $\texttt{(w}_\texttt{i}\texttt{,1,1,1)}$ is not allowed.

Clearly, such specifications are of little use. What we are interested in, are systems that behave in a *reasonable* way. We formalize this notion next.

Definition 2.9 *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 matches the length of its input to the length of its output. Note that, without the third requirement, it is not of much use. The shared memories $\mathcal{S}^U$, $\mathcal{S}^{NC}$ and $\mathcal{S}^{ND}$ are length-preserving, $\mathcal{S}^{\emptyset}$ is not.

If the second condition holds, a memory specification is *complete*. Completeness is the requirement that a memory specification should not be able to reject any program as long as it is syntactically correct with respect to the interface. Among the above shared memories, $\mathcal{S}^U$ and $\mathcal{S}^{ND}$ are complete. 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.

The third condition says that any permutation used as a mapping from the instructions of the input to the responses of the output should respect the response relation of the interface. There are some subtle points to note. First, it requires that the output stream, $\mathbf{q}$, be at least as long as the input stream, $\mathbf{p}$; it could be greater (a problem which is taken care of by the requirement of length-preservation). 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 arguably weak requirements are favored for the sake of generality. Both of the shared memories $\mathcal{S}^{ND}$ and $\mathcal{S}^{NC}$ satisfy this third property; $\mathcal{S}^{\emptyset}$ and $\mathcal{S}^U$ do not.

Definition 2.10 *A restriction of a memory specification $\mathfrak{S}$ to the set $\Sigma' \subseteq \Sigma^{\mathfrak{S}}$ is the memory specification $\mathfrak{S}[\Sigma']$ with*

1. $\mathfrak{F}^{\mathfrak{S}[\Sigma']} = \mathfrak{F}^{\mathfrak{S}}[\Sigma']$
2. $\lambda^{\mathfrak{S}[\Sigma']} = \lambda^{\mathfrak{S}} \cap (((\mathcal{I}^{\mathfrak{F}^{\mathfrak{S}}[\Sigma']})^* \times (\mathcal{O}^{\mathfrak{F}^{\mathfrak{S}}[\Sigma']})^*) \times \mathbf{Perm})$

Definition 2.11 *A memory specification $\mathfrak{S}$ is called finite if*

1. $\Sigma^{\mathfrak{S}}$ *is finite.*
2. $\mu^{\mathfrak{S}}$ *is a rational relation.*
3. $\lambda^{\mathfrak{S}}((\mathcal{I}^{\mathfrak{S}})^* \times (\mathcal{O}^{\mathfrak{S}})^*)$ *is bounded.*

If $\mathfrak{S}$ is not finite, then it is *infinite.* The finiteness of a memory specification is related to its implementability by a finite state machine. The first two cases should be obvious; without their being satisfied, a memory specification cannot be realized by a finite state machine. The third condition appeals to a characteristic of the user, which we will analyze in the next subsection.

We shall conclude this subsection with the definition of parameterized instances. Let $\mathcal{S}$ be a shared memory specification. Let P, A, D be subsets of $\mathbb{N}$. Let $\Sigma' = \mathcal{I}' \cup \mathcal{O}'$, where

$$\mathcal{I}' = \{\mathtt{r_i}\} \times P \times A \ \cup \ \{\mathtt{w_i}\} \times P \times A \times D$$

$$\mathcal{O}' = \{\mathtt{r_o}, \mathtt{w_o}\} \times P \times A \times D$$

Then, $\mathcal{S}[\Sigma']$ is called a *parameterized instance*. If $\mathcal{S}$, P, A, D are all finite, $\mathcal{S}[\Sigma']$ is called a *finite* parameterized instance, or *finite instance* for short. We shall usually identify a parameterized instance with the tuple $\langle P, A, D\rangle$, denoted by $\mathcal{P}_{\langle P,A,D\rangle}$, or simply $\mathcal{P}$ when no confusion is likely to occur.

2.4.2 Implementation

Typically, we expect the memory specification to be used for defining shared memory models. That is, we are not really concerned about how a memory specification can be realized; it is to define all correct (allowed) input/output pairs. Any formalization, as long as it is mathematically sound, can be used to define the set of allowed pairs. An implementation, on the other hand, should be the mathematical description of something realizable. It is a machine that receives instructions, which it processes, and that generates responses. So, instead of the "static" definition of a specification, the implementation is necessarily "dynamic." We believe that a transducer captures this notion of dynamism as it helps us distinguish the input and output of a finite-state machine. Before proceeding on to the formal definitions, there are a few observations to make.

First of all, it should be obvious that since we are dealing with finite-state machines, the permutation used in the specification to map instructions to their responses is not adequate; we can only have finitely many input and output symbols. Our suggestion is to use a finite set of *colors* as a tag for each instruction and response. But this seems to introduce a new problem: how do we define the mapping once the color set is set? Before answering this question, let us move on to the next observation.

The user (per Fig. 2.2) is also a finite-state machine whose specifics we ignore. But its finiteness is crucial to our argument. When the user issues an instruction, it must have a certain mechanism to tell which response it receives actually corresponds to that instruction; this is especially true if both the user and the memory operate in the presence of pending or incomplete instructions. Let us assume that i_1 is an instruction that the user issued and the response r_1 is the symbol that the memory generated for i_1. When the user receives r_1 from the memory, it should be able to match it with i_1 without waiting for other responses. Furthermore, once i_1 and r_1 are paired by the user, they should remain paired; a future sequence of instructions and responses should not alter the once committed matchings. Since the user is modeled as a finite machine, it can retain only a finite amount of information about past input; most likely, it will only keep track of the *pending* instructions, instructions that have not received a response from the memory yet. These ideas are the basis for requiring implementations to be *immediate* and *tabular*, formalized below.

Once we assert that the user behaves in the aforementioned manner, we can tackle the relation between color sets and permutations. For any color set, we can assume that there is a certain interpretation function which defines a permutation when given two sequences of colors. The combination of the color set with this interpretation is called a *coloring*. Besides obeying the properties of the previous paragraph, we leave the interpretation, called *conversion function*, unspecified. We prove that any such coloring is equivalent to a certain canonical coloring which helps us do away with arbitrary colorings and work thereafter with the canonical coloring. This, in our view, is an important result: we are not assuming anything more than the finiteness of the user to obtain this result and removes, as we hope, a possible attack due to the arbitrariness of the canonical coloring, which could have been treated as an ad-hoc solution.[8]

We have previously talked about the universality of memories: they should not be allowed to stop operation or output when the user has still some pending instructions for which it expects responses. Once the implementation is defined as

[8] For instance, in [38], Hojati et al. actually used this canonical coloring without rigor, as a matter of fact.

a finite state machine, this property can be specified easily: at any reachable state of the implementation, there must always be at least one path to an accepting state such that no further input is read. We think that this is a basic requirement of any memory implementation, hence this is part of the definition of an implementation and not some property that it may or may not satisfy. We now proceed to formalize these ideas.

Let a *coloring* $\mathfrak{C}$ be the tuple $\langle C, \varphi \rangle$, where C is the *color* set and φ is the *conversion* function $(C^2)^* \rightarrow \mathbf{Perm}$. That is, a coloring defines a subset of $\mathbf{Perm}$ whose elements are in a correspondence with strings over pairs of colors. Of course, C could be chosen identical to $\mathbb{N}$, $dom(\varphi)$ restricted to $\bigcup_{k \in \mathbb{N}} (\mathbf{Perm}_k)^2$ and $\varphi(\mathbf{n}, \mathbf{m})$ defined to be the composition of $\mathbf{n}$ and $\mathbf{m}$ both of which are treated as permutations, to define all of $\mathbf{Perm}$. A coloring $\mathfrak{C}$ is *finite* if $C^{\mathfrak{C}}$ is finite.

We are now ready to define the principal mathematical structure used for memory implementations.

Definition 2.12 *A responder $\mathcal{M}$ over $\langle \mathcal{I}, \mathcal{O} \rangle$ with coloring scheme $\mathfrak{C}$ is a transducer $\langle \mathcal{I} \times C, \mathcal{O} \times C, Q^{\mathcal{M}}, q_0^{\mathcal{M}}, F^{\mathcal{M}}, E^{\mathcal{M}} \rangle$ such that for all $((\mathbf{p}, \mathbf{n}), (\mathbf{q}, \mathbf{m})) \in \tau^{\mathcal{M}}$, we have $|\mathbf{n}| = |\mathbf{m}| < |\mathbf{p}|, |\mathbf{q}|$, and $\varphi^{\mathfrak{C}}(\mathbf{n}, \mathbf{m}) \in \mathbf{Perm}_R$, for some $R \subseteq [|\mathbf{p}|]$. We further require that for all $q \in Q^{\mathcal{M}}$, if q is reachable from $q_0^{\mathcal{M}}$, then there is a path from q to some $r \in F^{\mathcal{M}}$, such that all the edges on this path are in $Q^{\mathcal{M}} \times ((\mathcal{O} \times (C \cup \{\varepsilon\})) \cup \{\varepsilon\}) \times Q^{\mathcal{M}}$. It is called length-preserving, if $|\mathbf{p}| = |\mathbf{q}|$ and $R = [|\mathbf{p}|]$.*

A responder $\mathcal{M}$ is *finite* if $\mathfrak{C}^{\mathcal{M}}$ is finite, and the sets $\mathcal{I}$, $\mathcal{O}$ are finite. Before getting into the specifics of colorings, we will need the following lemma.

Lemma 2.1 *Let T be a finite, length preserving transducer. Then, there exists a number N_T, such that for any accepting run of T, the difference between the number of input symbols read and the number of output symbols generated cannot exceed N_T.*

Proof (Lemma 2.1): Let N_T be the cardinality of the state space of T. Assume that T has an accepting run $\mathbf{r} = q_0 a_1 \cdots q_m$ such that at some state q_k in $\mathbf{r}$, we have $|\mathbf{s}^i| > |\mathbf{s}^o| + N_T$,[9] where $\mathbf{s}$ is the prefix of $\mathbf{r}$ that ends at q_k. Let $\mathbf{t}$ be the suffix $a_{k+1} \cdots q_m$. Then there necessarily exist some repeating states in $\mathbf{s}$. Find the first such repeating state, say q_i and its last occurence in $\mathbf{s}$, and construct a new run where the part between the two occurences is removed. The remaining run completed with $\mathbf{t}$ is still an accepting run of T. Now, in the removed portion, if there are no nonempty labeled edges, or the number input symbols is equal to the number of output symbols, repeat the procedure, as there are still at least N_T input symbols. In the eventual case where there is an unequal number of input and output symbols removed, the resulting run, although in the language of the transducer, cannot have the same number of input and output symbols, thereby contradicting the assumption that the transducer was length preserving.

□

Let φ be the conversion function of a finite, length preserving responder. An output symbol (o_i, m_i) of a run $\mathbf{r}$ is *immediate* if it is mapped by φ to an input symbol that preceded it in the label of $\mathbf{r}$. Formally, let (o_i, m_i) be in $\mathbf{r}^o$ for some run $\mathbf{r}$. Let s_k be the state just following (o_i, m_i) in $\mathbf{r}$. Then, it is immediate if it is mapped to an input symbol found in the label of the run $s_0 a_0 \cdots s_k$; that is, there is $a_j = (p_l, n_l) \in (\mathcal{I} \times C)$, $j < k$ and $\varphi(\sharp_2(\mathbf{r}^i), \sharp_2(\mathbf{r}^o))(l) = i$. A run is immediate if all its output symbols are. A finite, length preserving responder is immediate if all its runs are. It follows from the definition of boundedness and Lemma 2.1 that an immediate responder is bounded (by $|Q^{\mathcal{M}}|$).

As an example, consider the case depicted in Fig. 2.3. Time is assumed to progress from left to right; i_1 would be the first symbol in the label, r_3 would be the last. The mapping given on the left is immediate, as each response is mapped to an instruction that precedes its temporally. The mapping on the right, however, is not immediate as it maps r_1 to i_3 where i_3 appears later in the temporal order than r_1. This would intuitively correspond to the case where a response is generated to an instruction that is yet to be read.

[9] Without loss of generality, we are assuming that there are more input symbols than output symbols. The other case is symmetric.

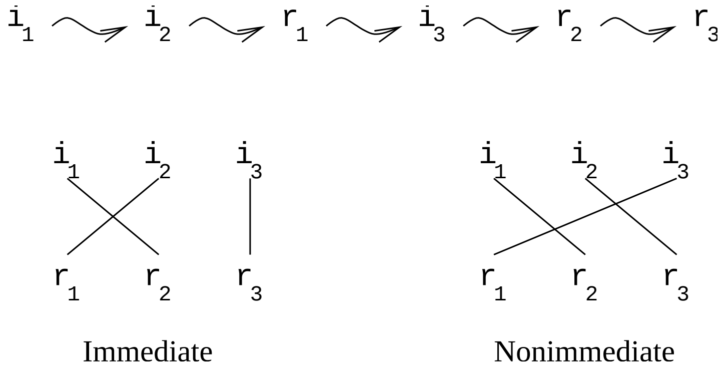


Figure 2.3. Two mappings for the same instruction/response stream pair. The above line represents the temporal ordering of the instructions and responses. The mappings give instances of mappings that are/are not immediate.

Let f be a function $C^* \times C \to C$. We call f a *template* for the conversion function φ of an immediate responder $\mathcal{M}$ if for all $(\mathbf{n}, \mathbf{m}) \in dom(\varphi)$, we have $\varphi^{-1}(\mathbf{n}, \mathbf{m})(k) = f(n_{i_1} \cdots n_{i_j}, m_k) + b_k$. The sequence $n_{i_1} \cdots n_{i_j}$ is the subword of $n_1 \cdots n_r$, $r \leq min(k + |Q^{\mathcal{M}}|, |\mathbf{n}|)$, such that all n_l with $\varphi(\mathbf{n}, \mathbf{m})(l) < k$ are removed and n_r is the color of the last input symbol read before the k^{th} output symbol whose color is m_k is generated. The number b_k is an offset equal to the number of symbols removed in the prefix $n_1 \cdots n_{i_{f(n_{i_1} \cdots n_{i_j}, m_k)}}$. Note that, the subword can be at most of length $|Q^{\mathcal{M}}|$. Intuitively, the template f, when given a sequence of colors $\mathbf{c}$ and a color c, returns the rank (the first letter of $\mathbf{c}$ being first) of the input symbol to which c is mapped. A conversion function for which a template exists is called *tabular*.

Again, we will give an example illustrating the tabularity of a mapping. In Fig. 2.4, we are giving not only the temporal ordering of instructions and responses as in Fig. 2.3, but also the color of each instruction and response. We assume that there are two different conversion functions, φ_1 and φ_2. For the first string, the mappings given by the two conversion functions are identical. However, when the first instruction (and its response) are removed, the mappings generated by φ_1 and φ_2 differ. The mapping on the left is due to φ_1 which is a tabular mapping. It can be seen that $\mathtt{d}_2$ is still mapped to $\mathtt{c}_3$. Note that, when the response with color

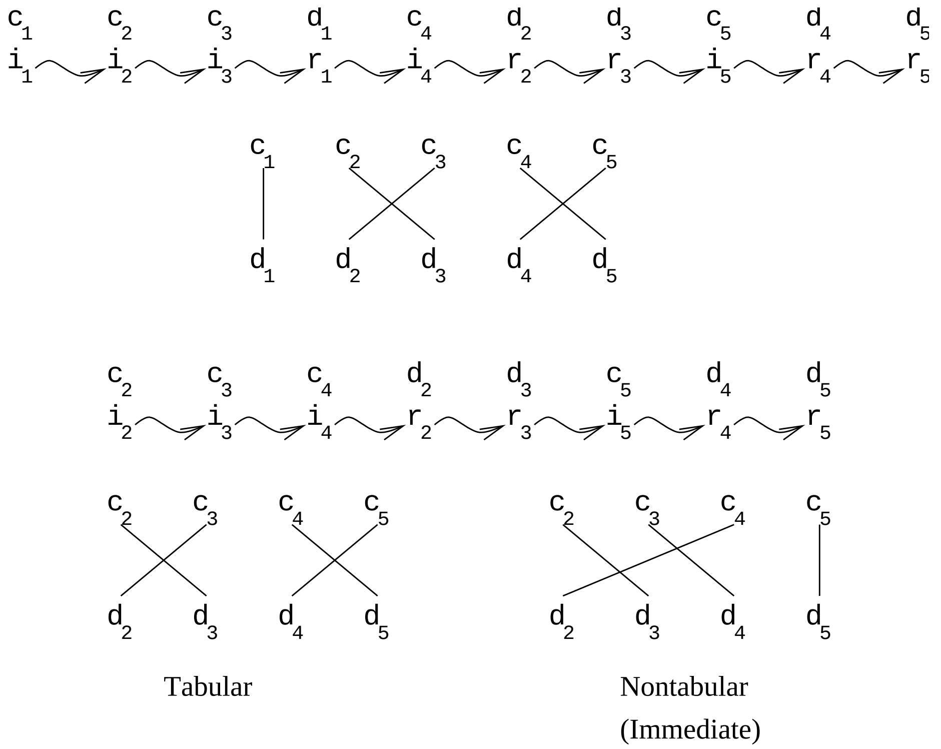


Figure 2.4. Two mappings, one tabular and the other not, have the same mapping for the first execution but they differ on the second.

$\mathtt{d}_2$ is generated in the second string, the input to the template function of φ_1 is the same, $(\mathtt{c}_2\mathtt{c}_3\mathtt{c}_4, \mathtt{d}_2)$, as the first string in which after generating $(\mathtt{r}_1,\mathtt{d}_1)$, $\mathtt{c}_1$ does not affect the remaining mappings. So it is expected that the remaining mappings remain the same under a tabular conversion function. However, the mapping on the right is different and hence φ_2 is not tabular. Intuitively, it can be said that how φ_2 maps a response depends not only on the set of pending instructions but also on the previous mapping history, a property which will possibly require a user with an infinite amount of storage.

Let C be a finite set of colors. Its elements will be denoted by $c_i, 0 < i \leq |C|$. Let $\mathbf{n}, \mathbf{m} \in C^*$. They are *compatible* if $\mathbf{n} \upharpoonright c_i = \mathbf{m} \upharpoonright c_i$, for all $c_i \in C$. Let $\mathbf{n}, \mathbf{m}$ be compatible. Then, the *normal* permutation from $\mathbf{n}$ to $\mathbf{m}$, $\widetilde{\eta}(\mathbf{n}, \mathbf{m})$, is defined as

$$\widetilde{\eta}(\mathbf{n}, \mathbf{m})(i) = j \text{ iff } (n_1 \cdots n_i) \upharpoonright n_i = (m_1 \cdots m_j) \upharpoonright m_j$$

That is, $\widetilde{\eta}(\mathbf{n},\mathbf{m})$ matches the occurrences of each color; the position of the first occurrence of c_1, if any, in $\mathbf{n}$ is mapped to the position of the first occurrence of c_1 in $\mathbf{m}$, and so on. That this mapping is well-defined follows from the definition of compatibility. A finite responder $\mathcal{M}$ for which $\varphi^{\mathcal{M}} = \widetilde{\eta}$ is said to be in *normal form.*

The pairs given in Fig. 2.5 give an example of compatible and non-compatible pairs of color strings. The one on the left is compatible as there is an equal number of occurrences of each color, $\mathtt{c}_1$ two times, $\mathtt{c}_2$ three times, $\mathtt{c}_3$ one time, in both strings. The one on the right, however, has two occurrences of $\mathtt{c}_1$ in one string and only one occurrence of $\mathtt{c}_1$ in the other. The figure also illustrates how the mapping would be done by the normal permutation $\widetilde{\eta}$ for the compatible pair.

For a label $\sigma = ((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m}))$ of a responder $\mathcal{M}$, define the *induced label*, $\widetilde{\sigma}$, to be $((\mathbf{p},\mathbf{q}),\mathbf{n}')$ such that $\mathbf{n}' \sim \varphi^{\mathcal{M}}(\mathbf{n},\mathbf{m})$. Define the *induced language* $\widetilde{\mathcal{L}}(\mathcal{M})$ of a responder $\mathcal{M}$ as

$$\widetilde{\mathcal{L}}(\mathcal{M}) = \{\widetilde{\sigma} \mid \sigma \in \tau^{\mathcal{M}}\}$$

Lemma 2.2 *Let $\mathcal{M}$ be an immediate responder, such that $\varphi^{\mathcal{M}}$ is tabular. Then, there exists an immediate responder $\mathcal{N}$ in normal form such that $\widetilde{\mathcal{L}}(\mathcal{M}) = \widetilde{\mathcal{L}}(\mathcal{N})$.*

Proof (Lemma 2.2): Let $\mathcal{M} = \langle \mathcal{I} \times C, \mathcal{O} \times C, Q, q_0, F, E\rangle$, $\mathfrak{C} = \langle C, \varphi\rangle$. We assume that $C = \{1,\ldots,m\}$. Let $D = \{1,\ldots,N\}$, where N is the bound $N_{\mathcal{M}}$ from Lemma 2.1. We assume that f is the template for φ. Let c_i, d_i range over

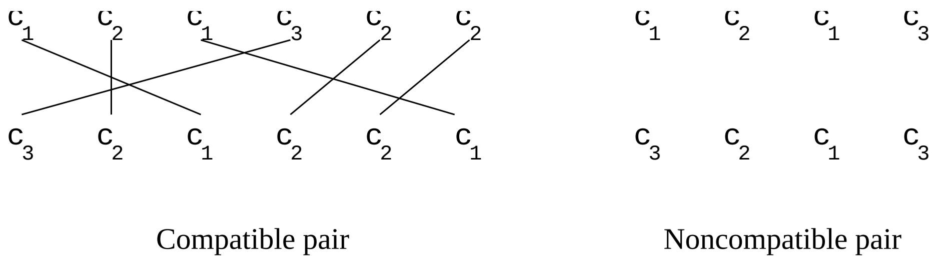


Figure 2.5. Two pairs of color strings, only one of which is compatible. For the compatible pair, we also provide the mapping given by the normal permutation, $\widetilde{\eta}$.

the elements of C and D, respectively. Define $\mathcal{N}$ over $\langle \mathcal{I}, \mathcal{O} \rangle$ with coloring scheme $\langle D, \widetilde{\eta} \rangle$ to be the transducer $\langle \mathcal{I} \times D, \mathcal{O} \times D, Q', q'_0, F', E' \rangle$ where

1. $Q' = Q \times \mathfrak{P}([N]) \times (C \times D)^{\leq N}$. An element of $\mathfrak{P}([N])$, a subset of $[N]$, will denote the colors of D which correspond to instructions whose responses have not been generated. $(C \times D)^{\leq N}$ is the set of all strings of length less than or equal to N over $(C \times D)$. It is used to encode the input to f whenever a response is generated by $\mathcal{M}$. It is also used to update the set of used colors of D.

2. $q'_0 = q_0 \times \emptyset \times \varepsilon$. We start from the initial state of $\mathcal{M}$ while no color in D is being used.

3. $F' = F \times \emptyset \times \varepsilon$. A state in Q' is final if its projection onto Q is in F and no color in D is being used; a condition that is satisfied only if there is no pending (colored) instruction.

4. $((q, U, \mathbf{s}), a, (q', U', \mathbf{s}')) \in E'$ if one of the following holds

   (a) $a = \varepsilon$ and $(q, \varepsilon, q') \in E$ and $U' = U$, $\mathbf{s}' = \mathbf{s}$.

   (b) $a = (p, d_i) \in \mathcal{I} \times D$, $d_i \notin U$, $(q, (p, c), q') \in E$, $U' = U \cup \{d_i\}$, $\mathbf{s}' = \mathbf{s} \cdot (c, d_i)$.

   (c) $a = (p, d_k) \in \mathcal{O} \times D$, $d_k \in U$, $(q, (p, c), q') \in E$, $\mathbf{s} = (c_1, d_1) \cdots (c_n, d_n)$, $f(c_1 \cdots c_n, c) = k$, $\mathbf{s}' = (c_1, d_1) \cdots (c_{k-1}, d_{k-1})(c_{k+1}, d_{k+1}) \cdots (c_n, d_n)$, $U' = U \setminus \{d_k\}$.

   So, $\mathcal{N}$ mimics $\mathcal{M}$ with some extra information about the combination of colors in C used as an input for f. The case (a) does not update the extra information in U or $\mathbf{s}$. In case (b), $\mathcal{M}$ inputs (p, c) while moving from q to q'. Hence, color $c \in C$ becomes part of the input to f for the next mapping of a response. Since f is arbitrary, its input might be any string in $C^{\leq N}$. To compensate this and keep the color sequences in D compatible, we need at most N different colors, hence the cardinality of D. So, we choose an arbitrary color $d \in D$ that is currently not in U (the set of used colors), and append (c, d) to $\mathbf{s}$. In case (c), $\mathcal{M}$ outputs (p, c) while moving from q to q'. Since f is tabular (and $\mathcal{M}$ is immediate), we can find to which instruction this (p, c) is mapped to. And since $\mathcal{N}$ is guaranteed to have a different color in D for each pending instruction, $\mathcal{N}$ can output p paired with the color of the instruction to which f maps (p, c). This also makes the input and output color sequences compatible.

Let $h_q : Q \rightarrow Q'$ be defined as

$$h_q(q) = \{(q, U, \mathbf{s}) \mid U \subseteq [N],\ \mathbf{s} \in (C \times D)^{\leq N}\}$$

Also, define h_l as

$$\begin{array}{rcl} h_l(\varepsilon) & = & \varepsilon \\ h_l((p, c)) & = & \{(p, d) \mid d \in D\}, \quad p \in \mathcal{I} \cup \mathcal{O}, c \in C \end{array}$$

Then, assume that $q_0 a_1 q_1 \ldots a_t q_t$ is an accepting run in $\mathcal{M}$. By the explanations above, it is easy to see that there exist q'_j, b_j, $1 \leq j \leq t$ such that $q'_0 b_1 q'_1 \ldots b_t q'_t$ is an accepting run of $\mathcal{N}$ where $q'_j \in h_q(q_j)$, $b_j \in h_l(a_j)$. Similarly, if $q'_0 b_1 q_1 \ldots b_t q_t$ is an accepting run of $\mathcal{N}$, there exist a_j, $1 \leq j \leq t$ such that $q_0 a_1 q'_1 \ldots a_t q'_t$ is an accepting run of $\mathcal{M}$, where $q'_j = h_q^{-1}(q_j)$, $a_j \in h_l^{-1}(b_j)$.

Combining all the previous arguments with the fact that $\mathcal{N}$ preserves the instruction to response mapping of $\mathcal{M}$, we conclude that their induced languages are equal; that is, $\widetilde{\mathcal{L}}(\mathcal{M}) = \widetilde{\mathcal{L}}(\mathcal{N})$.

□

Finally, we have to establish the link between specifications and implementations.

Definition 2.13 *An immediate responder $\mathcal{M}$ with a tabular conversion function is said to implement a (finite) memory specification $\mathfrak{S}$ if*

1. $\Sigma^{\mathfrak{S}} = \mathcal{I}^{\mathcal{M}} \cup \mathcal{O}^{\mathcal{M}}$

2. $\widetilde{\mathcal{L}}(\mathcal{M}) \subseteq \lambda^{\mathfrak{S}}$.

By virtue of Lemma 2.2 and the arguments done at the beginning of this section pertaining to the finiteness of the "user," from now on, by an implementation, we will mean an immediate responder in normal form.

The implementation is *exact* if the inclusion in the second condition above is an equality. Note that, for any implementation we can define a specification for which the implementation is exact. Such a specification will be called the *natural specification* of an implementation. Henceforth, when we talk about properness, completeness, etc. of an implementation, we will be referring to its natural specification.

2.5 Formalization in Action - Shared Memory Models

In this section, we will illustrate one use of our formalization. It is no secret that when we were defining specifications, we had a specific application in mind: that of formalizing shared memory models. In the first subsection, we will give a formal definition of one such shared memory model, sequential consistency, due to [43]. We have to also note that we will be exclusively dealing with some theoretical aspects of sequential consistency in the following chapters. In the next section, we will demonstrate how a finite instance of the lazy caching protocol [8] can be modeled as an implementation.

2.5.1 Sequential Consistency as a Specification

A shared memory model is a restriction on the output streams that can be generated for a given input stream. Hence, it is actually a predicate on the input-output relation $\lambda^{\mathcal{S}}$ of a given shared memory specification $\mathcal{S}$. In the following, we will describe sequential consistency, chosen for its simplicity and for its being accepted as a basic model by the research community.

There is no single formulation of sequential consistency, although the informal definition thereof is well-known[43]:

> [A memory system is sequentially consistent] if the result of any execution is the same as if the operations of all the processors were executed

> in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

The general approach is first to characterize the *correctness* of an execution, and then to require each execution that can be generated by the design under inspection to satisfy this correctness criterion. The formalizations usually differ in how they represent an *execution*, hence in how they define the property to be satisfied per execution. [references and examples: graph-based, rule-based, trace-based] It should not matter which formulation is picked as long as the input/output relations are equivalent. What follows is one such formulation.

For a string $\mathbf{q} = q_1 q_2 \cdots q_n$, let $\widehat{\mathbf{q}} = (q_1, 1)(q_2, 2) \cdots (q_n, n)$, be the *augmented string* of $\mathbf{q}$. For $\widehat{\mathbf{q}}$, let $(q_i, i) \prec_t^{\mathbf{q}} (q_j, j)$ if and only if $i < j$. $\prec_t^{\mathbf{q}}$ is called the temporal ordering induced by $\mathbf{q}$. When there is no confusion, we will abuse the notation and write $q_i \prec_t^{\mathbf{q}} q_j$ whenever $(q_i, i) \prec_t^{\mathbf{q}} (q_j, j)$. The *initial state* of a shared memory is a mapping $img(\alpha) \rightarrow img(\delta)$. For the sake of clarity, we will usually assume that the image of the initial state is simply $\{0\}$.

Definition 2.14 *Let $\sigma = ((\mathbf{p}, \mathbf{q}), \mathbf{n}) \in ((\mathcal{I}^{\mathcal{RW}})^* \times (\mathcal{O}^{\mathcal{RW}})^*) \times \mathbf{Perm}$. Then, σ is interleaved-sequential at the initial state ι if $|\mathbf{p}| = |\mathbf{q}| = |\mathbf{n}|$ and there exists a total ordering $\prec_l^{\mathbf{q}}$ over $\{j \mid 1 \leq j \leq |\mathbf{q}|\}$ such that*

1. *For any $i \in [|p|]$, if $\eta(i) = j$, then $\rho^{\mathcal{RW}}(q_j, p_i)$, where $\eta \sim \mathbf{n}$.*

2. *Let $j < k$, $\pi(p_j) = \pi(p_k)$ and i, l be such that $i = \eta(j)$, $l = \eta(k)$. Then, $i \prec_l^{\mathbf{q}} l$ (local input stream order, or input/program order).*

3. *If $q_j \in R$, then either*

$$(\neg \exists k \leq |\mathbf{q}| \,.\, q_k \in W \wedge \alpha(q_k) = \alpha(q_j) \wedge k \prec_l^{\mathbf{q}} j) \wedge$$
$$\delta(q_j) = \iota(\alpha(q_j))$$

or,

$$\exists k \leq |\mathbf{q}| \,.\, q_k \in W \wedge \alpha(q_k) = \alpha(q_j) \wedge$$
$$k \prec_l^{\mathbf{q}} j \wedge \delta(q_j) = \delta(q_k) \wedge$$
$$(\neg \exists l \leq |\mathbf{q}| \,.\, q_l \in W \wedge \alpha(q_l) = \alpha(q_j) \wedge k \prec_l^{\mathbf{q}} l \prec_l^{\mathbf{q}} j).$$

The ordering $\prec_l^{\mathbf{q}}$ is called the logical order of $\mathbf{q}$ in σ.

A label σ of a run of an implementation is interleaved-sequential (i-s, for short) if $\widetilde{\sigma}$ is i-s.

So a program and an execution are i-s if they are of the same length, each instruction generates a response conforming to the output relation of the interface, and it is possible to rearrange the responses of the execution such that the per processor order of the input is preserved and each read returns the value written by the most recent write to the same address with respect to the logical ordering or the initial value in the absence of any such write.

Given this definition of i-s, we can now define sequential consistency as a shared memory (specification).

Definition 2.15 *Sequential consistency is the shared memory* $\mathcal{S}^{SC} = \langle \mathcal{RW}, \lambda^{SC} \rangle$*, where* $\sigma \in \lambda^{SC}$ *if and only if* σ *is i-s.*

It is also possible to talk about the sequential consistency of a shared memory $\mathcal{S}$ when the relation defined by $\lambda^{\mathcal{S}}$ is a *nice* subset of the relation defined by λ^{SC}.

Definition 2.16 *A shared memory specification* $\mathcal{S}$ *is sequentially consistent if* $\mathcal{S}$ *is proper*[10] *and all elements of* $\lambda^{\mathcal{S}}$ *are i-s.*

The above definition has a novelty: for the first time, to the best of our knowledge, a specification is required to be *proper* to be sequentially consistent. As argued previously, when only the execution, the output stream, is used to define sequential consistency, or any memory model for that matter, it is impossible to characterize a proper specification.

2.5.2 Sequential Consistency of Implementation

The sequential consistency of an implementation does not follow from the sequential consistency of its specification. We have required the specification to be

[10] Actually, a complete specification is enough as the remaining constraints are implied by i-s elements. We should also note that all previous definitions of sequential consistency ignored properness.

proper for sequential consistency. This property, unfortunately, does not simply carry over to implementations as in its definition we only require an inclusion relation; certain programs might not produce outputs in the implementation. Therefore, in the case of an implementation, we will have to use its natural specification.

Definition 2.17 *An implementation is sequentially consistent if its natural specification is sequentially consistent.*

2.6 Implementation of Lazy Caching

Sequential consistency and high performance always seem to be perceived at opposite ends. If a sequentially consistent memory is designed, it is believed that it comes at the expense of possible interleavings of memory accesses as memory accesses would require stalling of the system to ensure that sequential consistency is preserved (see, for instance, [32]). However, the ease of programming for sequentially consistent memories makes them difficult to do away with. In this section, we will formalize the lazy caching protocol of [8], which provides a sequentially consistent memory with an improved management of memory accesses, i.e., less blocking. We will use an informal yet intuitive presentation, much like the original description of [8]. As can be verified, the only substantial difference between our definition and that of [8] is the use of colors, as should be expected.

The original transition structure of the lazy caching protocol is given in Table 2.1.

The memory system is a collection of p processors. Processor i, $i \in [p]$, has an output queue, Out_i, and input queue, In_i, and a cache, C_i. The queues are assumed to be unbounded.

The following operations are defined for queues:

- $append(queue, item)$ adds $item$ as the last entry in $queue$.
- $head(queue)$ returns the first entry from $queue$.
- $tail(queue)$ returns the result of removing $head(queue)$ from $queue$.

Table 2.1. The original transition structure of a lazy caching memory, as given in [8].

	The observable transitions	
$(\mathtt{r_i},i,a)$ ::		
$\mathit{handshake}[i] = \mathit{null}$	$\rightarrow$	$\mathit{handshake}[i] := (\mathtt{r_i},i,a)$;
$(\mathtt{r_o},i,a,d)$ ::		
$\mathit{handshake}[i] = (\mathtt{r_i},i,a)$	$\rightarrow$	$\mathit{handshake}[i] := \mathit{null}$;
$\wedge\ C_i[a] = d$		
$\wedge\ \mathit{is\_empty}(\mathit{Out}_i)$		
$\wedge\ \mathit{no\_star}(\mathit{In}_i)$		
$(\mathtt{w_i},i,a,d)$ ::		
$\mathit{handshake}[i] = \mathit{null}$	$\rightarrow$	$\mathit{handshake}[i] := (\mathtt{w_i},i,a,d)$;
$(\mathtt{w_o},i,a,d)$ ::		
$\mathit{handshake}[i] = (\mathtt{w_i},i,a,d)$	$\rightarrow$	$\mathit{handshake}[i] := \mathit{null}$; $\mathit{Out}_i := \mathit{append}(\mathit{Out}_i, (d,a))$;
	The internal transitions	
$\mathit{MW}_i(d,a)$ ::		
$\mathit{head}(\mathit{Out}_i) = (d,a)$	$\rightarrow$	$\mathit{Mem}[a] := d$; $\mathit{Out}_i := \mathit{tail}(\mathit{Out}_i)$; $k \neq i \Rightarrow \mathit{In}_k := \mathit{append}(\mathit{In}_k, (d,a))$; $\mathit{In}_i := \mathit{append}(\mathit{In}_i, (d,a,*))$;
$\mathit{MR}_i(d,a)$ ::		
$\mathit{Mem}[a] = d$	$\rightarrow$	$\mathit{In}_i := \mathit{append}(\mathit{In}_i, (d,a))$;
$\mathit{CU}_i(d,a)$ ::		
$\neg \mathit{is\_empty}(\mathit{In}_i)$	$\rightarrow$	$\mathit{In}_i := \mathit{tail}(\mathit{In}_i)$; $C_i := \mathit{update}(C_i, \mathit{data}(\mathit{head}(\mathit{In}_i)))$;
CI_i ::		
	$\rightarrow$	$C_i := \mathit{restrict}(C_i)$;

The predicates for queues are:

- $no\_star(q_i)$ returns true only if the queue q_i has no entry of the form $(d, a, *)$.
- $is\_empty(queue)$ returns true only if $queue$ is empty, that is, has no non-empty entries.

The arrays, which may also be considered as partial functions, have the following meaning:

- $handshake$, from $[p]$ to $\mathcal{I}^{\mathcal{RW}}$, keeps track of the pending instruction for each processor. The protocol lets processor i generate a response if the response matches $handshake[i]$, as in $(\mathtt{r_o}, i, a, d)$ can be generated only if $handshake[i]$ is $(\mathtt{r_i}, i, a)$.
- Mem, from $\mathbb{N}$ to $\mathbb{N}$, represents the global shared memory. $Mem[a]$ gives the content of address a.
- C_i, from $\mathbb{N}$ to $\mathbb{N}$, represents the local memory of processor i. $C_i[a]$ gives the value of address a as viewed by processor i.

Other miscellaneous functions are:

- $data((d, a, *))$ or $data((d, a))$ return the tuple (d, a).
- $update(cache, (data, address))$ returns a new array which agrees with $cache$ on all indices except for $address$ for which it returns $data$.
- $restrict(cache)$ returns a new array that agrees with $cache$ on all indices for which the new array is defined. That is, it is a restriction of $cache$ on a subset of the domain of $cache$.

A pictorial view of the protocol is given in Fig. 2.6.

Processor i can issue an instruction only if it has no pending instruction, that is, $handshake[i] = null$.[11] If the issued instruction is a write, that is, $(\mathtt{w_i}, i, a, d)$

[11] We have slightly changed the definition of [8] in that an instruction cannot be removed before it is serviced.

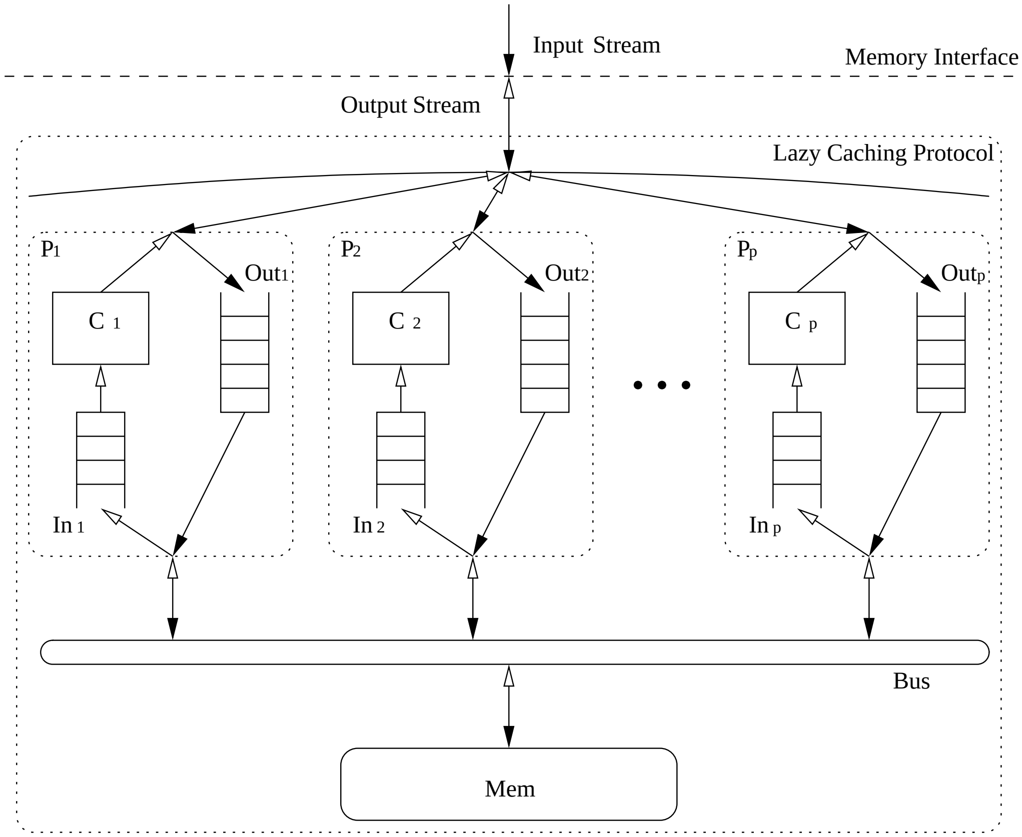


Figure 2.6. The lazy caching protocol.

for some a and d, its response can be generated at any time. When such a response, $(\mathtt{w_o}, i, a, d)$, is generated, the entry in $handshake[i]$ is cleared and the data, address pair (d, a) is placed in the queue Out_i. The response, $(\mathtt{r_o}, i, a, d)$, for a read instruction, $(\mathtt{r_i}, i, a)$, can be generated only if there are no entries of the form $(d, a', *)$ in In_i, Out_i is empty and the cache C_i is defined for address a, which intuitively means that the line for address a is in the cache C_i. These guards imply that all the preceding write instructions issued by processor i must have reached the global memory Mem (the emptiness of Out_i) and all the preceding local writes have reached the cache C_i. When the response, $(\mathtt{r_o}, i, a, d)$, is generated, the processor enables the issuing of the next instruction by resetting $handshake[i]$ (to $null$).

Besides the external or observable actions, the protocol also has some internal transitions. $MW_i(d,a)$ represents the updating of the global memory by processor i whose head entry of Out_i is (d,a). As the global memory is updated, the entry (d,a) is inserted into each In_k, for $k \neq i$. Into In_i, $(d,a,*)$ is inserted to tag it as a local write.

$CU_i(d,a)$ is the updating of the cache of processor i, C_i, by which the storing of the data value d in the address a is modelled.

As a nondeterministic measure to compensate for the abstraction of caching policies, two transitions, $MR_i(d,a)$ and CI_i, are used. The former places an entry (d,a) into In_i to model the case where a read miss occurs in cache C_i and the value is requested from the main (global) memory. CI_i undefines the mapping C_i for arbitrary, possibly 0, addresses, modelling the flushing of cache lines.

Since this definition uses unbounded queues In_i and Out_i, it clearly cannot be implemented as it is by a finite-state machine. To that end, we have to add guards which would check the fullness of a (finite sized) queue before enabling a transition which would require appending as a consequent action. A similar transition structure is given in Table 2.2 which describes the implementation of a finite instance of the lazy caching protocol, denoted by $LC(s_{in}, s_{out})$, where each In_i queue has capacity s_in_i, each Out_i has capacity s_out_i, and $s_{in} = \max_i\{s\_in_i\}$, $s_{out} = \max_i\{s\_out_i\}$ and where by finite, we mean that the address space A and the data space D are also finite.

As we have explained above, there can be at most one pending instruction per processor. This means that a color set of cardinality $|[p]|$ will be enough. Let the color set, C, be $\{c_1, \ldots, c_p\}$. It is not hard to see that the *handshake* array maps each instruction to its response. Since we will be using colors to achieve this mapping, *handshake* becomes redundant, so we will not use it. Instead, we will employ a subset of C, called U, of used colors. Each processor will have a unique associated color, c_i for processor i. During the operation of the implementation, processor i will have a pending instruction if and only if $c_i \in U$.

Table 2.2. The transition structure of an implementation modelling an instance of a lazy caching protocol.

The observable transitions

$$\begin{array}{lll}
\multicolumn{3}{l}{((\texttt{r}_{\texttt{i}}\texttt{,}\,i\,\texttt{,}\,a\texttt{)}, c_i) ::} \\
\quad c_i \notin U & \rightarrow & U := U \cup \{c_i\};
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{((\texttt{r}_{\texttt{o}}\texttt{,}\,i\,\texttt{,}\,a\,\texttt{,}\,d\texttt{)}, c_i) ::} \\
\quad c_i \in U & \rightarrow & U := U \setminus \{c_i\}; \\
\quad \wedge\ C_i[a] = d & & \\
\quad \wedge\ \mathit{is\_empty}(\mathit{Out}_i) & & \\
\quad \wedge\ \mathit{no\_star}(\mathit{In}_i) & &
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{((\texttt{w}_{\texttt{i}}\texttt{,}\,i\,\texttt{,}\,a\,\texttt{,}\,d\texttt{)}, c_i) ::} \\
\quad c_i \notin U & \rightarrow & U := U \cup \{c_i\};
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{((\texttt{w}_{\texttt{o}}\texttt{,}\,i\,\texttt{,}\,a\,\texttt{,}\,d\texttt{)}, c_i) ::} \\
\quad c_i \in U & \rightarrow & U := U \setminus \{c_i\}; \\
\quad \wedge\ \mathit{not\_full}(\mathit{Out}_i, s\_out_i) & & \mathit{Out}_i := \mathit{append}(\mathit{Out}_i, (d, a));
\end{array}$$

The internal transitions

$$\begin{array}{lll}
\multicolumn{3}{l}{\varepsilon ::} \\
\quad \mathit{head}(\mathit{Out}_i) = (d, a) & \rightarrow & \mathit{Mem}[a] := d; \\
\quad \wedge\ \forall k.\mathit{not\_full}(\mathit{In}_k, s\_in_k) & & \mathit{Out}_i := \mathit{tail}(\mathit{Out}_i); \\
 & & k \neq i \Rightarrow \mathit{In}_k := \mathit{append}(\mathit{In}_k, (d, a)); \\
 & & \mathit{In}_i := \mathit{append}(\mathit{In}_i, (d, a, *));
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\varepsilon ::} \\
\quad \mathit{Mem}[a] = d & \rightarrow & \mathit{In}_i := \mathit{append}(\mathit{In}_i, (d, a)); \\
\quad \wedge\ \mathit{not\_full}(\mathit{In}_i, s\_in_i) & & \\
\multicolumn{3}{l}{\varepsilon ::} \\
\quad \neg \mathit{is\_empty}(\mathit{In}_i) & \rightarrow & \mathit{In}_i := \mathit{tail}(\mathit{In}_i); \\
 & & C_i := \mathit{update}(C_i, \mathit{data}(\mathit{head}(\mathit{In}_i)));
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\varepsilon ::} \\
 & \rightarrow & C_i := \mathit{restrict}(C_i);
\end{array}$$

To take care of the finiteness of In_i and Out_i, we define a new predicate for queues, $not\_full(queue, size)$ which returns true only if $queue$ has less than $size$ entries.

We represent all the internal transitions by the empty string, ε. The language of the implementation must be a subset of Σ^*, so any other label is abstracted in this manner.

Since the transition structure given in Table 2.2 is almost identical to the original description barring the adjustments listed above, it deserves no further explanation.

Observe that, for any $i \in [p]$, In_i can never be stuck with some entries; the guard for removing an element from In_i requires only that In_i be non-empty (the guard for CU_i). This in turn implies that the guards for committing an instruction in the Out_j queue to the memory, for any $j \in [p]$, cannot remain false. Therefore, as long as a certain fairness constraint is added to the run, which prevents the cache from repeatedly flushing the address for which there is a read instruction pending, the lazy caching protocol is free from deadlock.[12] This also means that, one requirement for the finite-state machine to become an implementation, that of the ability to reach an accepting state from any reachable state without expecting further input, is satisfied.

It is not hard to see that the machine is length-preserving; each instruction generates exactly one response. Because each response is mapped to an instruction preceding the response in time, the machine is immediate. Adding to the fact that the machine will always generate compatible pairs and the mapping is in normal form, we conclude that the description given for a finite instance of the lazy caching protocol is indeed an implementation.

2.7 Summary

In this chapter, we introduced a new formalization for shared memories. The main motivation was to overcome the shortcomings of previous work in this area, as explained in Section 2.1. We have formally stated that a shared memory model

[12] For a detailed analysis, see [8].

is basically a *complete* specification. We have defined a canonical implementation and showed its generality which only depends on the finiteness of the memory implementation and the party interacting with this memory implementation. We have also used this formalization to define sequential consistency (as a specification) and the lazy caching protocol as an implementation. The next chapter will build on these results.

CHAPTER 3

VERIFICATION OF SHARED MEMORIES AS A LANGUAGE INCLUSION PROBLEM

In this chapter, building on the results of the previous chapter, we will reformulate the formal verification of a shared memory model. We will demonstrate our approach by proving the sequential consistency of the lazy caching protocol. We will demonstrate the use of memory model machines: their use enables one to transform the formal verification of a shared memory model into a regular language inclusion problem.

3.1 Introduction

Since Lamport's definition, there have been many attempts at characterizing theoretical aspects of SC (e.g., [12, 13, 36, 37, 48, 57]) and developing verification approaches for shared memory protocols with SC as the specification (e.g., [16, 35, 50]).[1] A memory system is usually modelled as a finite-state machine which is sequentially consistent if all the words in its language satisfy a certain property, which we called *interleaved-sequentiality*[2] in the previous chapter. Interleaved-sequentiality requires that the completed instructions (responses) can be ordered[3] in such a way that the per processor order is preserved and any read on any address returns the value of the most recent write (with respect to the logical order) to that address, or the initial value when no write to that address precedes the read.

[1] Hereafter referred to as "SC verification."

[2] Other names used previously are *serial* or *linear*. Due to the other well-established meanings of these words, we propose interleaved-sequential, which we also hope reflects better the intended meaning.

[3] This order is called the *logical order*.

There are different ways to characterize this property. An approach based on trace theory defines an execution, a word over Σ, to be a trace in a partially commutative monoid. A trace is interleaved-sequential if it is in a certain equivalence class. We call this the *trace based approach* to modelling SC.[4]

Unfortunately, the trace based approach to modelling SC introduces a number of problems described as mentioned in Chapter 1 and Appendix A. A related concern is that while two notable efforts ([16] and [53] being their latest publications) have given algorithms for SC verification, neither have formally contrasted the languages that correspond to the definition of SC they employ to the language of *(finitely) implementable sequentially consistent systems.*

We have already defined sequential consistency as a shared memory specification in the preceding chapter. In this chapter, we define an infinite family of finite-state implementations each of which implements sequential consistency. These implementations, called the *SC* machines, are actually modified versions of the serial memory, which has been previously used as an operational definition of sequential consistency (for instance, [1]). We describe the serial memory, which we call *serial machine* in Section 3.2.

In Section 3.3, we define the parameterized class of *SC* machines. We show that if the language of a memory implementation is contained in the language of some *SC* machine then the memory implementation is SC. Furthermore, this check can be accomplished using *regular language containment.* This is the first formulation of SC that we are aware of in which the problem of checking a given finite automaton against the SC specification is reduced to that of checking regular language inclusion with the members of a parameterized family of regular languages: if the containment succeeds for one instance of the parameters, we can assert that the protocol is SC.[5]

[4] See, for instance, [12].

[5] It is worth noting that Hojati et al. [38] formalized a certain verification effort based on refinement as a language inclusion problem, but that approach was implementation dependent as is usually true for refinement proofs in general.

In Section 3.4, we compare the family of SC machines against the most recent work on SC verification. The comparison is based on the class of languages each work defines, and the program/execution pairs each work deems interleaved-sequential.

We argue that TSC which is the class of languages defined to be sequentially consistent in the trace-based approach, is a superset of SC, which is the class of sequentially consistent languages. We show that there are languages in TSC consisting of program/execution pairs that are *illegal* from our point of view, as well as from the point of view of anyone who can view both the program executed as well as the execution that the program generated. This, we believe, may result in wrong claims being made about SC. In particular, the well-known undecidability proof of [12] does not apply to the domain of memory systems as defined in this work as the language used in that paper is in $TSC \setminus SC$.[6]

We further argue that SC_m, the class of languages recognized by SC machines, is contained within FSC, the class of sequentially consistent languages that are recognized by finite state machines, which is contained in SC and finally which is contained in TSC. We are then able to present the following results with respect to this hierarchy:

- We can show that there are finitely implementable SC languages that are not captured by [16], [53], or even SC_m that we propose.
- We show that both [16] and [53] contain almost no member of SC_m; their algorithms would be inconclusive in proving almost all members of SC_m to be SC.

In Section 3.5, we employ the SC machine to prove that restricted[7] finite instances of the lazy caching protocol, are sequentially consistent. The proof is accomplished by language containment: for each $LC_{n_q}(s_{in}, s_{out})$ (see Section 3.5),

[6] The language used in the final reduction is not *complete*, a property that we require to be satisfied by any sequentially consistent system.

[7] An enabled transition cannot be postponed for an unbounded number of transitions.

there is an $SC(J, K)$ machine, where the language of the former is a subset of that of the latter. We also argue that the unbounded case has an unfairness, a property which is precisely the reason why the language containment argument for any SC machine does not work. However, we also define another infinite-state machine, which basically is an SC machine with unbounded processor and commit queues (see Section 3.3) and argue that its language is a superset of the language of any $LC(s_{in}, s_{out})$.

In Section 3.6, we briefly point to the fact that the method carried throughout this chapter for defining the family of implementations for sequential consistency and the satisfaction of sequential consistency as a language inclusion problem has, indeed, nothing specific to sequential consistency; the method could be employed for any shared memory model and implementation as long as the formalization of Chapter 2 is followed.

Finally, in Section 3.7, we present a summary of the results of this chapter.

3.2 The Serial Machine

As a first step, we will define the *serial machine* as a specification, which in fact has an exact implementation. The serial machine is an operational model for sequential consistency. Its diagram is given in Fig. 3.1. It is a machine that does not behave “as if ...” [43] but operates as such.

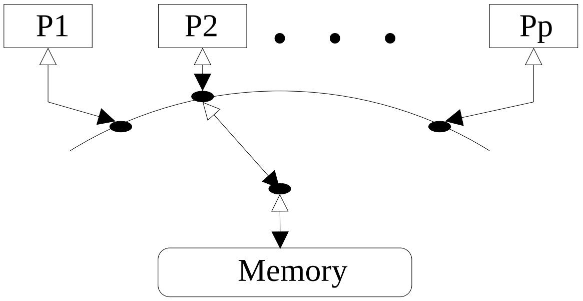


Figure 3.1. A serial machine with p processors

Basically, there is a single shared memory, one input port and one output port. The machine nondeterministically accepts through its input port one of the instructions each processor tries to issue. If the instruction is a write of a value, v, to an address a, then the value of a in the memory is updated to reflect the change and the instruction completes. If the instruction is a read of an address, then the value in the memory is passed to the processor which issued this instruction, and the instruction completes. The instruction is completed and the result is passed to the issuing processor through the output port before another instruction is accepted.

Formally, the definition of the serial machine is as follows:

Definition 3.1 *The serial machine, $\mathcal{S}^{SM} = \langle \mathcal{RW}, \lambda^{SM} \rangle$, is the shared memory specification with $\sigma = ((\mathbf{p}, \mathbf{q}), \mathbf{n}) \in \lambda^{SM}$ if and only if $\mathbf{n}$ is the identity permutation of $\mathbf{Perm}_{|\mathbf{p}|}$ and $\prec_t^{\mathbf{q}}$ is also a logical order for $\mathbf{q}$ in σ[8].*

We introduced the serial machine as an operational model for sequential consistency. We now establish the link between them.

Lemma 3.1 *Let $((\mathbf{p}, \mathbf{q}), \mathbf{n})$ be i-s. Then, there exist two permutations $\eta_1, \eta_2 \in Perm_{|\mathbf{p}|}$ such that $((\mathbf{p}', \mathbf{q}'), \mathbf{n}') \in \lambda^{SM}$, where $\mathbf{n}'$ represents the identity permutation in $\mathbf{Perm}_{|\mathbf{p}|}$, $\mathbf{p}' = p_{\eta_1^{-1}(1)} \cdots p_{\eta_1^{-1}(|\mathbf{p}|)}$, and $\mathbf{q}' = q_{\eta_2(1)} \cdots p_{\eta_2|\mathbf{q}|}$.*

Proof (Lemma 3.1): Since $((\mathbf{p}, \mathbf{q}), \mathbf{n})$ is i-s, there is a logical ordering, $\prec_l^{\mathbf{q}}$, of $\mathbf{q}$ in σ. Define $\eta_2(i) = j$ if and only if q_i is the j^{th} element in the logical order. To have a consistent temporal ordering for $\mathbf{p}'$, set $\eta_1 = (\eta_2(\eta(i)))^{-1}$, where $\eta \sim \mathbf{n}$.

□

Example 2 *Let us analyze the following i-s element $((\mathbf{p}, \mathbf{q}), \mathbf{n})$ of λ^{SC}, where*

$$\begin{array}{rcl}
\mathbf{p} & = & \mathtt{(r_i,1,1)\ (w_i,1,1,1)\ (w_i,2,1,2)} \\
\mathbf{q} & = & \mathtt{(w_o,1,1,1)\ (r_o,1,1,2)\ (w_o,2,1,2)} \\
\mathbf{n} & = & \mathtt{2\ 1\ 3}
\end{array}$$

[8] $q_i \prec_t^{\mathbf{q}} q_j$ if and only if $i \prec_l^{\mathbf{q}} j$.

The logical ordering of $\mathbf{q}$ *is* $\mathtt{3} \prec_l^{\mathbf{q}} \mathtt{2} \prec_l^{\mathbf{q}} \mathtt{1}$*. According to the proof of Lemma 3.1, we have* $\eta_2 \sim \mathtt{3\ 2\ 1}$ *and* $\eta_1 \sim ((\mathtt{3\ 2\ 1})(\mathtt{2\ 1\ 3}))^{-1} = \mathtt{3\ 1\ 2}$*. Using these permutations, we get*

$$\begin{aligned}
\mathbf{p}' &= (\mathtt{w_i},\mathtt{2},\mathtt{1},\mathtt{2})\ (\mathtt{r_i},\mathtt{1},\mathtt{1})\ (\mathtt{w_i},\mathtt{1},\mathtt{1},\mathtt{1}) \\
\mathbf{q}' &= (\mathtt{w_o},\mathtt{2},\mathtt{1},\mathtt{2})\ (\mathtt{r_o},\mathtt{1},\mathtt{1},\mathtt{2})\ (\mathtt{w_o},\mathtt{1},\mathtt{1},\mathtt{1}) \\
\mathbf{n}' &= \mathtt{1\ 2\ 3}
\end{aligned}$$

Since $\mathbf{n}'$ *is the identity permutation, and the temporal and logical orders for* $\mathbf{q}$ *in* $\sigma' = ((\mathbf{p}', \mathbf{q}'), \mathbf{n}')$ *coincide, we conclude that* σ' *is indeed in* λ^{SM}*.*

Lemma 3.1 implies that for any element $\sigma = ((\mathbf{p}, \mathbf{q}), \mathbf{n})$ of λ^{SC}, there is at least one element $\sigma' = ((\mathbf{p}', \mathbf{q}'), \mathbf{n}')$ in λ^{SM} such that $\mathbf{q}$ rearranged according to its logical order is the same as $\mathbf{q}'$. So, in a sense, $\mathcal{S}^{SM}$ is a *logically* complete specification with respect to sequential consistency. However, this fact is of little use if we want to formulate the checking of sequential consistency as a language inclusion problem.

There seem to be two main shortcomings in $\mathcal{S}^{SM}$. One is its inability to generate arbitrary interleavings for instructions; if instruction i_1 temporally precedes instruction i_2, even if i_1 and i_2 belong to different processors, arbitrary interleavings between the two is not possible. Consider the following case. Processor P_1 is to issue $(\mathtt{r_i},\mathtt{1},\mathtt{1})$ next, whereas processor P_2 has $(\mathtt{w_i},\mathtt{1},\mathtt{1},\mathtt{3})$ as its next instruction. Even though, say, $(\mathtt{r_i},\mathtt{1},\mathtt{1})$ precedes $(\mathtt{w_i},\mathtt{1},\mathtt{1},\mathtt{3})$ in the input stream, a sequentially consistent implementation can commit $(\mathtt{w_i},\mathtt{1},\mathtt{1},\mathtt{3})$ first and then return $(\mathtt{r_o},\mathtt{1},\mathtt{1},\mathtt{3})$ for the read instruction. This is not possible in $\mathcal{S}^{SM}$; $(\mathtt{r_i},\mathtt{1},\mathtt{1})$ precedes $(\mathtt{w_i},\mathtt{1},\mathtt{1},\mathtt{3})$, which means that the read instruction will read the value written before $(\mathtt{w_i},\mathtt{1},\mathtt{1},\mathtt{3})$.

A similar problem arises when an output symbol is generated. As we have seen in Lemma 3.1, an output symbol cannot be generated after another output symbol it precedes in logical order. This means that once $\mathbf{p}$ is fixed, $\mathbf{q}$ is also fixed, whereas in λ^{SC}, for each $\mathbf{p}$, there are at least $|\mathbf{q}|!$ different $\mathbf{q}'$ and $\mathbf{n}'$ pairs such that $((\mathbf{p}, \mathbf{q}'), \mathbf{n}') \in \lambda^{SC}$.

It should be clear that a finite-state machine cannot generate all such $\mathbf{q}'$ for a given $\mathbf{p}$, but we can use finite approximations; for a given $\mathbf{p}$, $\mathcal{S}^{SM}$ has one $\mathbf{q}$ (and $\mathbf{n}$), ideally there are at least $|\mathbf{q}|!$ number of $\mathbf{q}$ and $\mathbf{n}$ pairs such that $((\mathbf{p},\mathbf{q}),\mathbf{n}) \in \lambda^{SC}$; the finite approximations introduced in the next section will have a number in between.

3.3 A Finite Approximation 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.

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 $P = [|P|]$, $A = [|A|]$, $C = [|C|]$. The diagram of $SC_{(\mathcal{P},C)}(j,k)$ is given in Fig. 3.2.

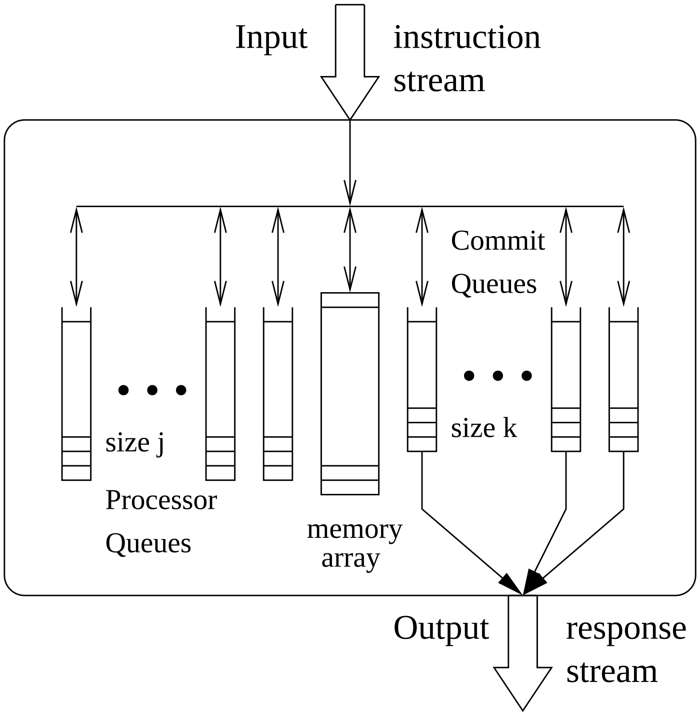


Figure 3.2. The diagram of $SC_{\mathcal{P},C}(j,k)$

The machine $SC_{(\mathcal{P},C)}(j,k)$ is defined as follows:

There are $|P|$ *processor* first-in first-out (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 (colored) 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 non-empty processor fifo queue. If this guard 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 $((\mathtt{r_o}, \pi(q), \alpha(q), mem(\alpha(q))), c)$ replaces the entry linked to (q, c) in the c^{th} commit queue. If $q \in W$, then the response $((\mathtt{w_o}, \pi(q), \alpha(q), \delta(q)), c)$ 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 that 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.

The pseudo-code of the machine is given in Fig. 3.3. For the sake of simplicity, the data structures and the standard routines for queues (insertion, copying the

```
INIT:
for all q $\in$ QP $\cup$ QC
  q.full:=FALSE;
  q.empty:=FALSE;
endfor
p_size:=j;
c_size:=k;
mem[a]:=$\iota$[a];
ready:=TRUE;
buf:=$\varepsilon$;

ITER:
RP:=$\{$q | q $\in$ QP, q.full=FALSE$\}$;
RC:=$\{$q | q $\in$ QC, q.full=FALSE$\}$;
OP:=$\{$q | q $\in$ QP, q.empty=FALSE$\}$;
OC:=$\{$q | q $\in$ QC, q.empty=FALSE AND (q.hd).fin=TRUE$\}$;
if ready AND buf=$\varepsilon$, then
  buf:=read(input);
  ready:=FALSE;
if buf$\neq\varepsilon$ AND QP[buf.q]$\in$RP AND QC[buf.c]$\in$RC, then
  ch:=$\{$INP$\}$;
if OC$\neq\emptyset$, then
  ch:=ch $\cup$ $\{$OUT$\}$;
if OP$\neq\emptyset$, then
  ch:=ch $\cup$ $\{$COM$\}$;
if ch=$\emptyset$, then TERMINATE;
c:=rnd(ch);
If c=INP, then
  id:=QC[buf.c].ins(buf); QP[buf.q].ins((buf,id)); ready:=TRUE;
  goto ITER;
if c=OUT, then
  q:=rnd(OC); output((q.pop).resp);
  goto ITER;
if c=COM, then
  q:=rnd(OP); ((p,c),id):=q.pop;
  if p $\in$ R, then
    p':=(r_o,$\pi$(p),$\alpha$(p),mem($\alpha$(p))); rep(id,(p',c));
  if p $\in$ W, then
    mem:=mem[<$\alpha$(p),$\delta$(p)>]; p':=(w_o,$\pi$(p),$\alpha$(p),$\delta$(p));
    rep(id,(p',c));
  goto ITER;
```

Figure 3.3. The pseudo-code for $SC_{\mathcal{P},C}(j,k)$.

head and popping) are not given; they should be clear from the explanations above.[9] The only "unorthodox" function is `rep` which takes a pointer and a data, and replaces the contents of the slot which the pointer points to (in a commit queue) with the data and sets a flag `fin` of that slot to `TRUE`.

We proceed to prove some properties of SC machines.

Lemma 3.2 *If there are some nonempty processor queues, the removal of any head entry from any of these queues is possible.*

Proof (Lemma 3.2): Since there is no additional guard (besides having at least one nonempty processor queue) which needs to be satisfied for the commit operation, any head entry of a nonempty processor queue can be committed, thereby updating the linked entry in the commit queue.

□

Lemma 3.3 *The $SC_{\mathcal{P},C}(j,k)$ machine never deadlocks, that is, it does not reach a state where either there is still unread input or there are some nonempty queues and none of the guards for any of the three operations is satisfied.*

Proof (Lemma 3.3): Since by Lemma 3.2, any head entry can be chosen to commit, we could empty the processor queues in l consecutive commit steps where l is the number of entries in the processor queues. Then, all entries of the commit queues, including the head entries, are ready to be output. That means that it is always possible to reach a state where all the processor and commit queues are empty. Since when these queues are empty, the guard for reading an input symbol is enabled, the reading of input cannot get stuck either. Therefore, the $SC_{\mathcal{P},C}(j,k)$ will not deadlock.

□

Lemma 3.4 *If* $((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m}))$ *is in* $\mathcal{L}(SC_{\mathcal{P},C}(j,k))$*, then* $\mathbf{n}$ *and* $\mathbf{m}$ *are compatible.*

[9] The mapping $mem[\langle a,d\rangle]$ agrees with the mapping mem on all addresses except for a where the new data value is given by d.

Proof (Lemma 3.4): Since nothing concerning the colors is done, an instruction and its committed form have the same color.

□

Lemma 3.5 *$SC_{\mathcal{P},C}(j,k)$ machine is an immediate responder in normal form.*

Proof (Lemma 3.5): The input order of instructions per color is preserved in the output due to the ordering imposed by the commit queues. That is, if (i,c) comes as the n^{th} instruction with color c in the input, its output (o,c), for some $o \in \mathcal{O}^{RW}$, will be the n^{th} response in the output with color c. Combining this with Lemma 3.4, we conclude that it is in normal form.

That it is immediate follows from the fact that for a response to an instruction to be output, the instruction first has to be read from the input.

□

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 $\bigcup_{j,k \in Nat} L(SC_{\mathcal{P},C}(j,k))$.

Lemma 3.6 *Let M be a shared memory implementation of instance $\mathcal{P}$, and let $\sigma = ((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m})) \in \tau^M$. Then, σ is i-s if and only if $\sigma \in L_{\mathcal{P},C^M}$.*

Proof (Lemma 3.6): (Only if) : Choose $j = k = |\mathbf{p}|$. With these values of j and k, all instructions can be held in the processor and commit queues without being forced to remove an element. Let $\prec_l^{\mathbf{q}}$ be the logical ordering of $\mathbf{q}$. Consider the following run of $SC_{\mathcal{P},C^M}(|\mathbf{p}|,|\mathbf{p}|)$. It first reads all the instructions into their respective queues. Then, consistent with the ordering dictated by $\prec_l^{\mathbf{q}}$, instructions are committed and the necessary changes are made in the commit queues. We recall that an implementation of a finite instance is assumed to be an immediate responder in normal form. That means that $\mathbf{n}$ and $\mathbf{m}$ are compatible. Therefore, as the final step, the commit queues are emptied according to the temporal ordering of $(\mathbf{q},\mathbf{m})$, in accordance with the mapping done by the normal permutation of M.

The other direction follows from the definition of i-s and the fact that the order of committing is the logical order of the output string.

□

Theorem 3.1 *A shared memory implementation M of instance $\mathcal{P}$ is sequentially consistent iff M is proper and the relation it realizes is in $L_{\mathcal{P},C^M}$.*

Proof (Theorem 3.1): Follows directly from the previous lemma and the definitions of $L_{\mathcal{P},C^M}$ and sequential consistency.

□

This theorem can also be used as an alternative yet equivalent definition of sequential consistency of implementations.

It should be clear that for finite values of j and k, the $SC_{\mathcal{P},C}(j,k)$ machine is finite iff $\mathcal{P}$ and C are finite.

Theorem 3.2 *Let M be an implementation of a finite instance $\mathcal{P}$. Then, M is sequentially consistent if M is complete and $L(M) \subseteq L(SC_{\mathcal{P},C^M}(j,k))$ holds for some $j,k \in \mathbb{N}$.*

Proof (Theorem 3.2): Follows from Thm. 3.1 and the definition of sequential consistency.

□

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 [15]). 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 the following.

Theorem 3.3 *Given an implementation M of a finite instance $\mathcal{P}$, it is decidable to check whether M is complete and has a language that is subset of some $SC_{\mathcal{P},C^M}(j,k)$, for some $j,k \in \mathbb{N}$.*

3.4 Related Work - A Comparison Based on Languages

Previous work on sequential consistency checking can be divided into two main classes: necessity and sufficiency.

The approach based on necessity tries to find necessary conditions for a sequentially consistent implementation. These conditions are then formalized and tested on a given implementation. Work in this vein include [22, 51]. As such, they are valuable as debugging tools; if at least one of the conditions fail for an implementation, it is concluded that the implementation is not sequentially consistent. However, if the conditions are satisfied, the effort is inconclusive; the implementation might still have runs that violate sequential consistency.

The approach based on sufficiency tries to prove that an implementation is sequentially consistent. As is true with all kinds of formal verification, there are those that are completely automatic [53, 16], those that are manual [18, 40] and those that lie in between [25, 19].

It is widely believed that a fully-automatic formal verification algorithm for any finite state memory implementation is undecidable.[10] Hence, the works of [53, 16], which are based on the same formalism of trace theory, try to characterize *realistic* or *practical* subsets whose membership problem is decidable.

The approach presented in this paper is an effort at the automatic formal verification for sufficiency and so it behooves to compare the recent results on that area with the results of this paper. The languages of the SC machines define yet another class of languages, which we will denote by SC_m. For the comparison

[10] For the undecidability result, refer to [12]. Due to the formalization used in this paper, we do not share this belief.

to make sense, we will translate the trace theoretical representations of the previous work to our setting.

3.4.1 The Work of Qadeer [53]

In this work, which revises the test automata approach of [50], a memory implementation is verified for sequential consistency through composition of the implementation with a collection of automata. The main assumption is that for any address and any two write instructions, w_1, w_2, to that address, w_1 precedes w_2 in issuing order if and only if the response to w_1 precedes the response to w_2 in the logical order. Another assumption, causality, states that if a value is read at an address, then that value is either written by an instruction or is the initial value.

In our formalization of the problem, the requirement that a read must return a value that is input into the system before the read is completed is stronger than causality, hence that assumption is already satisfied by any system claimed to be sequentially consistent. However, the first assumption about the logical ordering per address is not present in our framework; it is an assumption which we are reluctant to make, as the aim of this work is to be as general as possible and not appeal to experience.

As we shall see below, this class of languages, which we shall denote by L_q, includes languages that are not sequentially consistent, that are not finitely implementable, and that are finitely implementable but not among SC_m.

3.4.2 The Work of Bingham, Condon and Hu [16]

This work, which seems to be the culminating point of a series of previous work, such as [17, 23], has some interesting observations about the undecidability work. They argue that there are two properties that need to be satisfied by any memory implementation, however have not been ruled out by previous work employing trace theory. The first one, prefix-closedness, stresses the fact that a memory should not wait for certain inputs to reach an accepting state. The second, prophetic inheritance, states that a read cannot return the value of a write that is yet to occur. As should be obvious, the exact same requirements are also present in our framework. Actually, we believe that in an execution based formalization, which [16, 53] certainly are, it is impossible to characterize these notions.

Consider the output sequence $\mathtt{(r_o,1,1,1)}$ $\mathtt{(w_o,2,1,1)}$ which does not belong to the class DSC defined in [16]. This output sequence could belong to an execution that is prophetic, to an execution that is not prefix-closed, or to an execution that is neither; the correct characterization depends on the input stream. In the following examples, time is assumed to progress from left to right; in each instance, the upper line corresponds to the input, the lower to the output. We also assume that the initial value of address a is 0.

Instance 1	:	$\mathtt{(r_i,1,1)}$		$\mathtt{(w_i,2,1,1)}$	
			$\mathtt{(r_o,1,1,1)}$		$\mathtt{(w_o,2,1,1)}$
Instance 2	:	$\mathtt{(w_i,2,1,1)}$		$\mathtt{(r_i,1,1)}$	
			$\mathtt{(r_o,1,1,1)}$		$\mathtt{(w_o,2,1,1)}$
Instance 3	:	$\mathtt{(w_i,2,1,1)}$	$\mathtt{(r_i,1,1)}$		
				$\mathtt{(r_o,1,1,1)}$	$\mathtt{(w_o,2,1,1)}$

The first instance corresponds to the prophetic inheritance; the read instruction returns a value that has not yet been input into the system. The second corresponds to an execution that is not prefix-closed; a response to a read instruction that has not been input is generated. Only the third instance corresponds to a case which can be considered intuitively correct. It should not come as a surprise, then, to note that only the third instance is allowed in our framework.

We will use L_{bch} to denote the class of languages that [16] defines.[11]

In the following subsections, we will compare these two classes and the class defined in this paper based on the program/execution pairs and languages each class admits.

[11] In [16], this class is called DSC which is a limit of all DSC_k, for $k \in \mathbb{N}$. All DSC_k are finitely verifiable for sequential consistency.

3.4.3 Admissible Program/Executions

For the comparison to make sense, we will translate the trace theoretical representations of [16, 53] to our setting. We will also assume that the values of $\langle P, A, D\rangle$ and C are fixed and refer to the language union of all $SC_{\mathcal{P},C}$ machines as $L(SC_m)$.

For the following, let $L(q)$, $L(bch)$ denote the unions of the languages of classes L_q, L_{bch}, respectively. With respect to the program/execution pairs each class has (Fig. 3.4(a)), only $L(SC_m)$ is exactly equal to the program/execution pairs that are interleaved-sequential (i-s, for short). Neither $L(q)$ nor $L(bch)$ include all possible i-s behaviors. Furthermore, these sets are mutually incomparable and both contain behaviors that are not i-s.[12]

As an example, consider the region A of Fig. 3.4(a) which corresponds to i-s executions which are not in $L(bch) \cup L(q)$. The following is such an execution, which is given along with its program:

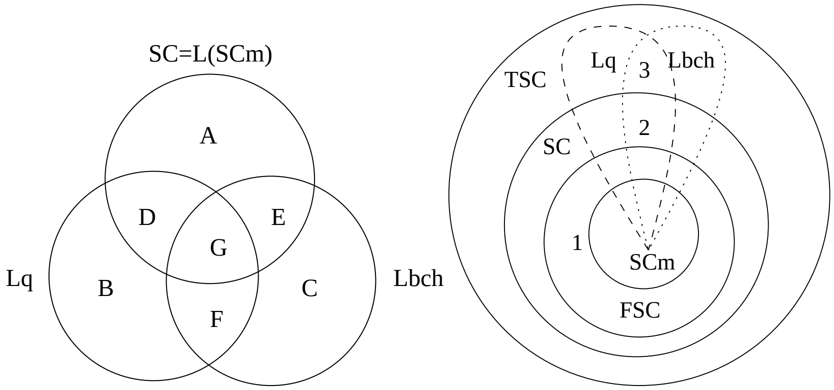
$$\begin{aligned} \text{Program} &: \texttt{(r}_\texttt{i}\texttt{,1,}a\texttt{)}\ \texttt{(w}_\texttt{i}\texttt{,1,}a\texttt{,1)}\ \texttt{(w}_\texttt{i}\texttt{,2,}a\texttt{,2)} \\ \text{Execution} &: \texttt{(r}_\texttt{o}\texttt{,1,}a\texttt{,2)}\ \texttt{(w}_\texttt{o}\texttt{,1,}a\texttt{,1)}\ \texttt{(w}_\texttt{o}\texttt{,2,}a\texttt{,2)} \end{aligned}$$

As can be seen, if the last write $\texttt{(w}_\texttt{o}\texttt{,2,}a\texttt{,2)}$ of processor 2 is logically ordered as the first response, the output stream becomes i-s. However, this execution is not in $L(bch)$ as the instruction $\texttt{(r}_\texttt{i}\texttt{,1,}a\texttt{)}$ receives a value that is not seen in the output so far. It is not in $L(q)$, as the temporal ordering of writes to address a is not the same as their logical ordering. We should note that this input/output pair can be generated by any $SC(j, k)$, for $j, k \geq 2$, regardless of the cardinality of the color set.

3.4.4 Admissible Languages

As for the classes of languages (Fig. 3.4(b)), the SC_m is included in the class of finitely implementable languages, denoted by FSC.

[12] Strictly speaking, this claim is true, as both will allow executions without any program. It might be argued that this is a mere technicality; we are ready to accept that point of view if the relation between programs and executions is made explicit.



(a) Word based

(b) Language based

Figure 3.4. Comparison diagrams.

Region 1 represents the class of languages that are finitely implementable but not covered by any of the L_q, L_{bch}, SC_m. As an example to such a language, consider the following set of input/output pairs[13]:

$$\begin{array}{rcl} \text{Input} & : & (\mathtt{w_i},\mathtt{1},a,\mathtt{1})\ (\mathtt{w_i},\mathtt{2},a,\mathtt{2})\ \cdot_1(b,-)^*\ \cdot_2(c,-)^*\ (\mathtt{r_i},\mathtt{2},a) \\ \text{Output} & : & \cdot_1(b,-)^*\ \cdot_2(c,-)^*\ (\mathtt{r_o},\mathtt{2},a,\mathtt{1})\ (\mathtt{w_o},\mathtt{1},a,\mathtt{1})\ (\mathtt{w_o},\mathtt{1},a,\mathtt{2}) \end{array}$$

The symbol $\cdot_i(x,-)$ is a wildcard denoting any symbol that has x in the address and issued/performed by processor i. The Kleene-$*$ has the usual meaning. We assume that except for the instructions on address a, every instruction is completed according to the input order. A finite state machine generating this input/output pair must only check for the addresses accessed so far. It will only detain two instructions on address a and complete the others in the order they appear as input which can be done using finite resources. If this set of input and output pairs is combined with the language of the serial machine, L_{SM}, we get a complete and sequentially consistent finite implementation that belongs to Region 1.

Region 2 represents the set of sequentially consistent languages, not finitely implementable, but included in both L_q and L_{bch}. Consider the following set of

[13] In this example, the relative ordering of output symbols to input symbols is not represented.

input/output pairs:

$$\begin{aligned} \text{Input} &: \texttt{(w}_\texttt{i}\texttt{,1,}a\texttt{,1)}\ \texttt{(r}_\texttt{i}\texttt{,2,}a\texttt{)}^k \\ \text{Output} &: \texttt{(w}_\texttt{o}\texttt{,1,}a\texttt{,1)}\ \texttt{(r}_\texttt{o}\texttt{,2,}a\texttt{,}d\texttt{)}^k \end{aligned}$$

We require that d be 1 only if $k = 2^n$ for some integer n, otherwise d is the initial value of a. The language formed by taking the union of the above with L_{SM} is sequentially consistent, although the part given above cannot be generated by any finite state machine.

Finally, for region 3, languages that are defined to be sequentially consistent under the formalism of trace theory (denoted by TSC in fig. 3.4(b)) and included in both L_q and L_{bch}, we can simply take the empty language. Since it has no violating executions, it is defined to be sequentially consistent! Other examples include $\mathcal{S}^{\emptyset}$ and $\mathcal{S}^{NC}$, defined in the previous chapter.

3.5 Lazy Caching Protocol and Sequential Consistency

The lazy caching protocol, described in the previous chapter, is notorious for the difficulty it causes when one tries to prove it sequentially consistent. There is a special issue of Distributed Computing, devoted exclusively to the proof that lazy caching is indeed sequentially consistent (see, for instance, [18, 34]). Almost all the methods used to prove lazy caching sequentially consistent are manual or highly dependent on the specifics of lazy caching, making those methods highly unlikely to be employed for the general case of shared memory verification. We would like to claim that for any finite implementation of the lazy caching protocol, there is an SC machine whose language is the superset of the implementation. Unfortunately, that is not true. But it might not be as bad as it sounds.

Let us first define a family of machines that define a *fair* implementation of the lazy caching protocol. Let $LC_{n_q}(s_{in}, s_{out})$ be the finite state machine whose language is included in the language of $LC(s_{in}, s_{out})$ with the following additional property.

For any run $\mathbf{r} = q_0 a_1 q_1 \ldots q_t$ and for any q_j, $j \in [t]$ and $j_c = min\{j + n_q, t\}$, the following holds for any $i \in P$:

1. If at q_j, Out_i is nonempty, then there is at least one MW_i transition between q_j and q_{j_c}.
2. If at q_j, In_i is nonempty, then there is at least one CU_i transition between q_j and q_{j_c}.

The relation between these LC_{n_q} machines and the SC machines is established by the following theorem.

Theorem 3.4 *For each $LC_{n_q}(s_{in}, s_{out})$, there are $SC_{\mathcal{P},P}(J,K)$ machines such that $\mathcal{L}(LC_{n_q}) \subset \mathcal{L}(SC_{\mathcal{P},P}(J,K))$.*

Proof (Theorem 3.4): We will construct, for any run $\mathbf{r}$ of $LC_{n_q}(s_{in}, s_{out})$, a run $\mathbf{r}'$ of $SC^a_{\mathcal{P},P}(J,K)$ such that the labels $(\mathbf{r}^i, \mathbf{r}^o)$ and $(\mathbf{r}'^i, \mathbf{r}'^o)$ are equal and $\mathcal{L}(SC^a_{\mathcal{P},P}(J,K)) \subset \mathcal{L}(SC_{\mathcal{P},P}(J,K))$.

Let us first define the following sets for convenience:

- $MW_i = \{MW_i(d,a) \mid d \in D, a \in A\}$
- $MW = \cup_{i \in P} MW_i$
- $WR_i = \{\texttt{((w}_\texttt{o}\texttt{,}i\texttt{,}a\texttt{,}d\texttt{),}i\texttt{)} \mid a \in A, d \in D\}$
- $WR = \cup_{i \in P} WR_i$
- $WI_i = \{\texttt{((w}_\texttt{i}\texttt{,}i\texttt{,}a\texttt{,}d\texttt{),}i\texttt{)} \mid a \in A, d \in D\}$
- $WI = \cup_{i \in P} WI_i$
- $RI_i = \{\texttt{((r}_\texttt{i}\texttt{,}i\texttt{,}a\texttt{),}i\texttt{)} \mid a \in A\}$
- $RI = \cup_{i \in P} RI_i$
- $RR_i = \{\texttt{((r}_\texttt{o}\texttt{,}i\texttt{,}a\texttt{,}d\texttt{),}i\texttt{)} \mid a \in A, d \in D\}$

- $CU_i = \{CU_i(d,a) \mid d \in D, a \in A\}$
- $MR_i = \{MR_i(d,a) \mid d \in D, a \in A\}$

Let $\mathbf{r} = q_0 a_1 q_1 \dots a_t q_t$ be a run of $LC_{n_q}(s_{in}, s_{out})$. Define $\widehat{\mathbf{r}} = q_0 b_1 q_1 \dots b_t q_t$ such that $b_j = a_j$ if $a_j \neq \varepsilon$; otherwise, b_j is the appropriate internal label (of the transition from q_{j-1} to q_j).

Let $n_w = |b_1 b_2 \dots b_t \restriction WR|$. That is, n_w gives the number of write responses (or equivalently, the number of write instructions) in $\widehat{\mathbf{r}}$ (or $\mathbf{r}$).

Let

- $ins\_o_p(n,m) = |b_n b_{n+1} \dots b_m \restriction WR_p|$
- $ins\_i_p(n,m) = |b_n b_{n+1} \dots b_m \restriction (MR_p \cup MW)|$
- $rmv\_o_p(n,m) = |b_n b_{n+1} \dots b_m \restriction MW_p|$
- $rmv\_i_p(n,m) = |b_n b_{n+1} \dots b_m \restriction CU_p|$

That is, $ins\_o_p(n,m)$ gives the number of insertions into Out_p from q_{n-1} to q_m in $\widehat{\mathbf{r}}$. Similarly, for the same interval, $ins\_i_p(n,m)$ gives the number of insertions into In_p; $rmv\_o_p(n,m)$, the number of removals of entries from Out_p; $rmv\_i_p(n,m)$, the number of removals of entries from In_p.

Let $mc : (WI \times [t]) \rightarrow [n_w]$ be such that $mc((\texttt{(w}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}, p), j) = k$, if $b_j = (\texttt{(w}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}, p))$, there is $l \in [t]$ such that $b_l = MW_p(d,a)$ and $|b_1 b_2 \dots b_l \restriction MW| = k$, and $ins\_o_p(1,j) - rmv\_o_p(1,j) = rmv\_o_p(j, l-1)$. That is, all removals from Out queues, hence updates of the global memory Mem, are uniquely identified by a number in $[n_w]$ which determines its temporal order in $\widehat{\mathbf{r}}$; $mc(wr, j) = k$ if the j^{th} label of $\widehat{\mathbf{r}}$ is wr and its data/address pair is the k^{th} update of Mem.

Let $pv : (Q^{LC_{n_q}(s_{in}, s_{out})} \times P) \rightarrow [n_w]$ be such that $pv(q_j, p) = k$, if there is $i \in [t]$ such that $b_i \in MR_p \cup MW$, $ins\_i_p(1,i) - rmv\_i_p(1,i) = rmv\_i_p(i,j)$, and $|b_1 b_2 \dots b_i \restriction MW| = k$. That is, for any $p \in P$ and $j \in [t]$, pv gives the number of writes seen at processor p till state q_j in $\widehat{\mathbf{r}}$.

Let $oldest(q_j) = min\{pv(q_j, i) \mid i \in P\}$; hence, it gives the number of writes the processor, which has updated its cache the least number of times, has seen.

Let $rc : (RR \times [t]) \rightarrow [n_w] \cup \{0\}$ be such that $rc(\texttt{((r}_\texttt{o}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{),}p), j) = k$, if $b_j = \texttt{((r}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{),}p)$, there is $l \in [t]$ such that $b_l = \texttt{((r}_\texttt{o}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{),}p)$, $|b_{j+1}b_{j+2} \dots b_{l-1} \upharpoonright RR_p| = 0$, and $pv(q_l, p) = k$. That is, $rc(rd, j) = k$ if rd is the response to the (read) instruction input during the transition from q_{j-1} to q_j and the processor view when rd is generated equals k.

Let $oo : ((RI \cup WI) \times [t]) \rightarrow [t]$ be such that $oo(i, j) = k$, if there is $l \in [t]$, $p \in P$ such that $i \in RI_p \cup WI_p$, $b_j = i$, $b_l \in WR_p \cup RR_p$, $|b_{j+1}b_{j+2} \dots b_{l-1} \upharpoonright (WR_p \cup RR_p)| = 0$, and $|b_1 \dots b_l \upharpoonright (WR_p \cup RR_p)| = k$. That is, $oo(i, j)$ gives the rank of the response for the instruction i input during the transition from q_{j-1} to q_j. The rank is one more than the number of responses generated in the prefix $q_0 b_1 q_1 \dots q_{l-1}$.

Let us now define $SC^a_{\mathcal{P},P}(J, K)$. It has the same structure as $SC_{\mathcal{P},P}(J, K)$ except that the entries of processor and commit queues have an extra parameter and there is a variable $no \in [t]$. An entry of a processor queue is of the form (i, k) where $i \in \mathcal{I}^{RW} \times P$ and $k \in [n_w] \cup \{0\}$. An entry of a commit queue is of the form (r, o), where $r \in \mathcal{O}^{RW} \times P$ and $o \in [t]$. Intuitively, k for (i, k) of a processor queue entry gives information about committing; hence k is called the *commit order* for (i, k). The value of o in (r, o) of a commit queue entry tells when it is ok to generate r as output; o is called the *output order* for (r, o).

Let the initial state of $SC^a_{\mathcal{P},P}(J, K)$ be the state where all queues are empty, ι agrees with Mem on A and $no = 1$. Starting from $j = 1$, the following steps are performed:

1. If $b_j = \texttt{((r}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{),}p)$, then insert $(b_j, rc(b_j, j))$ into $Proc_p$[14].

2. If $b_j = \texttt{((w}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{),}p)$, then insert $(b_j, wc(b_j, j))$ into $Proc_p$.

3. Let $s_{min} = oldest(q_j)$. Let k_{min} be the minimum among the commit orders of the head entries of all nonempty processor queues. If $s_{min} < k_{min}$, go to next step. Otherwise, that is, if $s_{min} \geq k_{min}$, there are two cases to consider (see Lemma 3.7).

[14] As a notational convenience, $Proc_p$ denotes the p^{th} processor queue.

Depending on the satisfied predicate, one of the following steps is chosen and performed:

(a) There is a head entry (i, k_{min}) where $i \in WI$. Commit this entry (and remove it from the processor queue). Then repeat the same for all head entries whose commit orders equal k_{min}. Repeat step 3.

(b) None of the entries with commit order k_{min} belong to a write instruction; that is, if (i, k_{min}) is a head entry, then $i \notin WI$. Commit all such (i, k_{min}). Repeat step 3.

4. Let o_{min} be the minimum output order of the head entries of all nonempty commit queues. Let (r, o_{min}) be the corresponding head entry. If $o_{min} = no$ and r is committed, generate r, increment no by 1 and repeat step 4. Otherwise, go to next step.

5. If $j < t$, increment j by 1 and go to step 1. Otherwise, terminate.

Observe that, if (i_1, k_1) is inserted into $Proc_p$ before (i_2, k_2), then $k_1 \leq k_2$. This is because for the same processor, rc and wc are nondecreasing for increasing values of $j \in [t]$. This implies that there cannot be an entry in the processor queues such that it is not a head entry and its commit order is less than k_{min} of step 3.

Lemma 3.7 *In step 3, either a head entry is (wr, k_{min}) where $wr \in WI$ or the entry (wr, k_{min}) with $mc(wr, j) = k_{min}$, for some $j \in [t]$, has been committed previously.*

Proof (Lemma 3.7): Assume the contrary. Then, it must be inserted into some $Proc_p$, $p \in P$, after at least another read instruction that has the same commit order (it cannot be larger, by the above observation). Let these entries be (wr, k_{min}) and (rd, k_{min}), for the write and read instructions, respectively. Note that, since they were issued by the same processor (p), the read instruction must have completed, its response must have been generated before the write instruction was input. Let $j < k < l$ be such that $b_j = rd$, $b_l = wr$ and b_k be the response for b_j (rd). By definition of rc, $rc(rd, j) = k_{min}$ implies that $|b_1 b_2 \ldots b_k \upharpoonright MW| = k_{min}$. But also, by the definition of wc, $wc(wr, l) = k_{min}$ implies $|a_1 a_2 \ldots a_k \ldots a_l \upharpoonright MW| < k_{min}$, which is a contradiction.

□

It remains to show that the sizes of queues need not be unbounded; that is, whenever a step (1 or 2) dictates an insertion into $Proc_p$ or $Commit_p$[15], there cannot be arbitrarily many entries in these queues.

Lemma 3.8 *If* $J, K \geq \lceil \frac{nq(s_{in}+s_{out})}{2} \rceil + 1$, *the queues will not overflow.*

Proof (Lemma 3.8): Let at state q_j, or the j^{th} iteration for $SC^{a}_{\mathcal{P},P}(J, K)$, the head entry in $Proc_p$ be $(\texttt{((r}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{)}\texttt{,}p\texttt{)}, k)$. This means that there is $m \in [t]$ such that $m \leq j$, $b_m = \texttt{((r}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{)}\texttt{,}p\texttt{)}$ and $rc(b_m, m) = k$. Note that, before the response for b_m is generated by $LC_{n_q}(s_{in}, s_{out})$, no other input from processor p can be inserted into $Proc_p$. So, let us also assume that $b_l = \texttt{((r}_\texttt{o}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}\texttt{,}p\texttt{)}$ is the response to b_m with $l \leq j$. Now note that, by the definition $LC_{n_q}(s_{in}, s_{out})$ and the fact that the write of $mc^{-1}(k)$ must be inserted into each In queue, at least right before q_{l-1} ($b_{l-1} = MW_p(d, a)$), for some $p \in P$, we have $oldest(q_{l_c}) \geq k$, where $l_c = min\{t, l+n_q s_{in}\}$. Between q_{l-1} and q_{l_c} (which covers q_j), there could be at most $\lceil \frac{n_q s_{in}}{2} \rceil$ entries inserted into $Proc_p$. Since when $oldest(q_{l_c}) \geq k$, $(\texttt{((r}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{)}\texttt{,}p\texttt{)}, k)$ can be committed, the size of the $Proc_p$ queue will never exceed $\lceil \frac{n_q s_{in}}{2} \rceil + 1$ when the head entry of $Proc_p$ is a read instruction.

Let us now assume that at q_j, the head entry of $Proc_p$ is $(\texttt{((w}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}\texttt{,}p\texttt{)}, k)$. This means that there is $m \in [t]$ such that $m \leq j$, $b_m = \texttt{((w}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}\texttt{,}p\texttt{)}$ and $mc(\texttt{((w}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}\texttt{,}p\texttt{)}) = k$. Similar to the previous argument, if the response to b_m has not been generated before q_j, there could be no other entries. So, assume that $b_l = \texttt{((w}_\texttt{o}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}\texttt{,}p\texttt{)}$ is the response to b_m with $l \leq j$. A response for a write can be generated by processor p if Out_p is not full. Assuming that prior to q_l, Out_p had $(s_{out} - 1)$ entries (this corresponds to the worst case), it would take at most $n_q(s_{in} + s_{out})$ transitions for all processors to see the effects of this

[15] The p^{th} commit queue.

write. That is, for $l_c = min\{t, l + n_q(s_{in} + s_{out})\}$, $oldest(q_{l_c}) \geq k$ and therefore by q_{l_c}, $(((\texttt{W}_\texttt{i}\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}\texttt{,}p), k)$ is guaranteed to have been committed. That would mean that there could be at most $\lceil \frac{n_q(s_{in}+s_{out})}{2} \rceil$ entries inserted after q_m into $Proc_p$ and before its head entry is committed.

Therefore, for processor queues, we must have $J \geq \lceil \frac{n_q(s_{in}+s_{out})}{2} \rceil + 1$.

Let us now consider the commit queues. First, observe that, if the next symbol to be output belongs to processor p and the instruction to which this symbol would be the response has not been input yet, $Commit_p$ must be empty and all the other commit queues can have at most one entry.

Let, at state q_j, the instruction whose response would be output next is inserted into $Commit_p$ (as well as $Proc_p$). This entry will remain in the queue as long as it is not committed. By previous arguments, we know that it will take at most $n_q(s_{in} + s_{out})$ transitions, which, in turn, means that at most $\lceil \frac{n_q(s_{in}+s_{out})}{2} \rceil$ new instructions can be input. Now, note that initially there was at most one entry in each $Commit$ queue. So, $K \geq \lceil \frac{n_q(s_{in}+s_{out})}{2} \rceil + 1$ is enough.

□

So far, we have not shown that the logical order of the run $\mathbf{r}'$ of $SC^a_{\mathcal{P},P}(J, K)$ corresponding to $\mathbf{r}$ is indeed compatible[16] with the logical order of $\mathbf{r}$. Now, observe that, mc is a bijection. Intuitively, the state of the global memory Mem changes every time an MW transition is done. So, there are $n_w + 1$ different[17] states of Mem, including the initial state and each write is uniquely associated with a state of Mem. These temporally ordered states of Mem actually give the logical order of the execution (for a proof, see [8]). Hence, by committing instructions according to commit orders given by mc, $SC^a_{\mathcal{P},P}(J, K)$ generates a compatible logical order. Since the instructions and responses are input and output in the order depicted by

[16] We are not saying *the same* logical order as for a given i-s program/execution pair, there might be more than one logical order.

[17] Strictly speaking, Mem can attain a state more than once, but that is immaterial to the present argument.

$\widehat{\mathbf{r}}$ (the former, by steps 1 and 2; the latter, by step 4 and the definition of oo), the label $(\mathbf{r}'^i, \mathbf{r}'^o)$ is the same as $(\mathbf{r}^i, \mathbf{r}^o)$.

Finally, note that $SC^a_{\mathcal{P},P}(J,K)$ actually defines but a subset of all the nondeterministic runs of $SC_{\mathcal{P},P}(J,K)$ as any transition enabled in the former is also enabled in the latter. Therefore, $\mathbf{r}' \in \mathcal{L}(SC_{\mathcal{P},P}(J,K))$ and we conclude that $\mathcal{L}(LC_{n_q}(s_{in}, s_{out})) \subset \mathcal{L}(SC_{\mathcal{P},P}(J,K)$.

□

What about the general case? There are two alternatives. First, we could prove it sequentially consistent, using (nonregular) language containment. Or, we could argue that the very fact that the language of $LC(s_{in}, s_{out})$ is not contained in the language of any $SC_{\mathcal{P},P}(j,k)$ is an undesirable property of the lazy caching protocol. Here, we will do both!

Define $SC^{\infty}_{\mathcal{P},C}$ as the machine that behaves like any $SC_{\mathcal{P},C}(j,k)$ except that all queues are unbounded.

Lemma 3.9 $\mathcal{L}(SC^{\infty}_{\mathcal{P},C}) = \mathcal{L}_{\mathcal{P},C}$.

Proof (Lemma 3.9): Let $((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m})) \in \mathcal{L}(SC^{\infty}_{\mathcal{P},C})$. Then, $((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m})) \in \mathcal{L}(SC_{\mathcal{P},C}(|\mathbf{p}|,|\mathbf{q}|))$. Hence, $\mathcal{L}(SC^{\infty}_{\mathcal{P},C}) \subseteq \mathcal{L}_{\mathcal{P},C}$.

Let $((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m})) \in \mathcal{L}_{\mathcal{P},C}$. Then, there exist infinitely many j and k such that $((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m})) \in \mathcal{L}(SC_{\mathcal{P},C}(j,k))$. Take one such j, k pair. But, it is obvious that $\mathcal{L}(SC_{\mathcal{P},C}(j,k)) \subset \mathcal{L}(SC^{\infty}_{\mathcal{P},C})$. Therefore, $((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m})) \in \mathcal{L}(SC^{\infty}_{\mathcal{P},C})$ and $\mathcal{L}_{\mathcal{P},C} \subseteq \mathcal{L}(SC^{\infty}_{\mathcal{P},C})$.

□

By a similar argument to the one we did in the proof of Theorem 3.4, it can be readily shown, then, that $\mathcal{L}(LC(s_{in}, s_{out})) \subset \mathcal{L}(SC^{\infty}_{\mathcal{P},P})$. That, together with the previous Lemma is enough to conclude that $LC(s_{in}, s_{out})$ is sequentially consistent.

Let us now consider the other option. In the lazy caching protocol, a processor might delay the synchronization of its local cache for an arbitrary period of time.

Think of the following set of program/execution pairs:

$$((\mathtt{w}_\mathtt{i},\mathtt{1},\mathtt{1},\mathtt{1}),\mathtt{1}) \qquad (\ ((\mathtt{r}_\mathtt{i},\mathtt{2},\mathtt{1}),\mathtt{2})\ ((\mathtt{r}_\mathtt{i},\mathtt{1},\mathtt{1}),\mathtt{1})\ )^*$$

$$((\mathtt{w}_\mathtt{o},\mathtt{1},\mathtt{1},\mathtt{1}),\mathtt{1}) \qquad (\ ((\mathtt{r}_\mathtt{o},\mathtt{2},\mathtt{1},\mathtt{0}),\mathtt{2})\ ((\mathtt{r}_\mathtt{o},\mathtt{1},\mathtt{1},\mathtt{1}),\mathtt{1})\ )^*$$

In any program/execution pair of this set, processor P_1 writes 1 to address 1, but processor P_2 ignores this value, at no point in time it updates its cache, and keeps reading the initial value for address 1, which is assumed to be 0. This set of program/execution pairs cannot be generated by any SC machine. Depending on the queue sizes, at some point, processor P_2 must see the updated value as the write to address 1 must be logically committed. However, these program/execution pairs can be generated by an implementation of the lazy caching protocol, as long as the In and Out queues have sizes greater than 0.

This is clearly an issue of fairness. A nondeterministic self-loop usually abstracts communication delays, or in general, uncertainties in temporal domain. However, the program/execution pair, given above, although adhering to the definition of sequential consistency, is not desired in an intuitively correct shared memory implementation. Adding a fairness constraint is not really relevant as we defined the problem in terms of finite programs/executions. If it were an infinite program execution, we could say that the processor P_2 eventually sees the updated value. It would be of little comfort for a programmer to know that processor P_2 would eventually see the updated value, when it has not seen it for two days of execution. There is actually a stronger notion, finitary fairness [11], that if a transition is enabled, it cannot remain enabled and not taken for longer than k transitions, where k is an unknown but finite number in $\mathbb{N}$. The bound n_q we employed could be seen as this k applied to finite strings (program/execution pairs).

Actually, the clash between an intuitively correct shared memory and sequentially consistent memory goes beyond the arguments presented here. Some problems, albeit somewhat less general than what could be, have been mentioned in [44]. The problem is more in proposing a better formulation than pointing out what is wrong with the definition of sequential consistency. For now, we would like to classify this issue as a future research topic.

3.6 The General Case for Shared Memory Models

Sequential consistency is but one among the myriad of memory models proposed: PRAM, coherence, weak ordering, TSO and PSO, Itanium, to name a few. Models like PRAM and coherence also are defined over the $\mathcal{RW}$ interface; that is, they are specifications for $\mathcal{RW}$. It is, however, a common practice to introduce new operations for accessing and changing the memory. The motivation behind a more complicated instruction set for memory accesses is that distinguishing between ordinary accesses from synchronizing accesses can result in performance improvement. Typically, the memory access instructions are categorized into weak and strong. The memory model usually guarantees a stricter ordering for strong instructions. For instance, in weak ordering, the program's execution with respect to the strong operations is sequentially consistent. The weak accesses, whose arbitrary interleavings per processor having no bearing on the correctness of the program, are usually ordered with respect to strong operations; ordering among weak operations is usually left unspecified.

We have attacked the problem of verifying the sequential consistency of a given implementation because sequential consistency is an important memory model, evidenced by the sheer volume of work done on it. Besides, the most recent verification results or theoretical work for shared memories usually concentrate on sequential consistency which makes it easier to compare the current work with similar research. However, what we have done in this chapter is in no way restricted to sequential consistency.

Let us take a step back and examine what exactly we did to come up with the *SC* machines.

1. We started with a certain memory interface: the rw-interface, $\mathcal{RW}$. This basically defined the communication primitives of the system.

2. With respect to this memory interface, we then described a shared memory model, sequential consistency. The approach was to specify what it meant to have a *correct* input stream/output stream pair, to require an implementation have only correct input stream/output stream pairs in its language and have at least one output stream for any possible input stream.

3. We observed that for any given input stream, generating all possible output streams was not possible by a finite-state machine. Instead, we showed that it is possible to finitely approximate the set of possible output streams for a given input stream. We defined a hierarchy of finite-state machines whose language limit is the same as the language of the memory model we were interested in, sequential consistency. Each machine, depending on the size of its processor and commit queues, can delay the committing of an instruction or the generation of a response.

Abstracting away the details pertaining to the specific problem we have dealt with, the above listing can be rewritten as follows:

1. Start with the required memory interface for the memory model.

2. Define what it means to have a correct output stream for an input stream.

3. Define the hierarchy of finite-state machines whose language limit will be equal to the memory model.

This three step outline can be applied to the verification of any finite-state implementation (in the sense introduced in this work), with respect to any memory model. The first step is trivial.

For the second step, the "how to define" or "how to formulate" part results in different formalisms. We have, in defining specifications, completely left what formalism to use to the specifier. The only requirement, which should be the requirement of any formalism, is that it be mathematically sound.

The third and final step, much like the previous step, might result in different formulations. We used queues; for some memory models, queues might prove to be useful; for others, multisets, partial orders, reorder buffers, etc. might be suitable. A suite of possible structures along with their use in defining operational models was given in [19]; the idea of using operational models to define and debug memory models go back to [25]. But the basic underlying idea remains the same. The operational description of each machine should be able to generate all possible input/output stream pairs up to a certain size. If the input exceeds that size, the part of the input that cannot be retained is assumed to have a certain logical execution, which is reflected by the state of the machine. This way, it is guaranteed to have all of the correct input/output stream pairs in the language of the hierarchy: just pick the machine whose size is enough to hold all of the input stream before being forced to generate an output symbol.

As for the verification of implementations, first check for completeness. If complete, proceed to finding a finite-state machine, whose language would be the superset of the language of the implementation, from the hierarchy.

3.7 Summary

In this chapter, we introduced a new approach to the formal verification of a shared memory implementation with respect to sequential consistency. Through the use of an updated alphabet (instructions and responses with colors), we were able to formulate the verification problem as a regular language inclusion problem. The SC machines whose language union gives all possible sequentially consistent program/execution pairs are quite intuitive.

We demonstrated the use of this formulation by proving the sequential consistency of the finite instances of the lazy caching protocol, restricted to bounded nondeterminism, the $LC_{n_q}(s_{in}, s_{out})$ machines, using (regular) language inclusion. We argue that there is no finite bound to nondeterministic self-loops, the $LC(s_{in}, s_{out})$ machine does possess an undesirable property which is precisely why its language is not contained in the language of any SC machine. However, we also showed, if desired, how to prove $LC(s_{in}, s_{out})$ sequentially consistent through a proof very similar to the one we did.

Finally, we claimed that the method proposed in this chapter did not depend on some specific properties of sequential consistency. Any shared memory implementation, modeled as an *implementation*, that is, as an immediate responder in normal form, can be verified for any shared memory model, which is (re)defined as the language union of an infinite family of implementations each of which approximates the memory model under consideration.

CHAPTER 4

SEQUENTIAL CONSISTENCY AND UNAMBIGUOUS EXECUTIONS

In this chapter, we will further look into the problem of checking sequential consistency for a finite-state system. We will propose a novel approach to checking the interleaved-sequentiality of an execution. We will prove the new problem, called the constraint satisfaction problem, to be equivalent to checking interleaved-sequentiality. Using this result, we will be able to obtain a decidability result for the set of unambiguous executions.

4.1 Introduction

As we have seen in Chapter 2 where we defined sequential consistency, a shared memory implementation is sequentially consistent if, besides other requirements, each of its executions are *correct*; that is, interleaved-sequential. Therefore, a related issue in the formal verification of sequential consistency is the correctness of a single execution.

The work presented in this chapter, starts from an analysis of a single concurrent execution which we coin the name *interleaved-sequentiality checking.* We first observe that each processor, through the responses to the instructions it issued, defines a set of possible orderings. Or, put in other words, it prohibits certain orderings. A concurrent execution is interleaved-sequential if the set of all prohibited orderings of all the processors is a proper subset of all possible (interleaved) orderings. Based on this observation, we prove that any unambiguous concurrent execution is interleaved-sequential if and only if a certain set of constraints on the ordering of the write events of the execution is satisfiable. Besides being an interesting formulation in itself, the constraint satisfaction problem can also be used to extract the set of *wrong* instructions which make a concurrent execution violate interleaved-sequentiality. We argue that this extraction in previous formulations of the problem is not as trivial as it is under ours. Using this characterization, we are able to obtain a finiteness result for sequential consistency checking: there exists a number k, a function of the address and data value spaces, such that if a shared memory implementation has noninterleaved-sequential and unambiguous executions, there exists at least one such unambiguous execution in which no state of the implementation is visited more than k times.

We should also note that we are trying to do away with one aspect common to both the debugging and proving camps: we are not trying to restrict the domain of our problem by assuming certain properties about the system. The assumptions such as location monotonicity or symmetry [35, 50], restriction on temporal order of writes per address [53], ruling out certain orderings of instruction completion or requiring in-order completion [16] are usually essential in deriving results. The theorems of this chapter do not rely on any assumption, although as we discuss in the closing section, similar assumptions would help alleviate the complexity of the solution.

It is worth adding that the existence of such assumptions is not uncalled for: the general case for the formal verification of a finite-state system for sequential consistency, when the formalization is based on execution only[1] as in [12, 16, 53] is undecidable; hence the efforts for carving out the largest subset of implementations for which checking sequential consistency is decidable. The finiteness and decidability result for unambiguous executions of this chapter shows that the undecidability result is implicitly linked to the ambiguity of the execution.

In the following section, we explain the notation used in this chapter, which is somewhat different than the notation used in the previous chapters. We also give the definitions of the structures used throughout this chapter. In Section 4.3, we state the original problem in our framework. In Section 4.4, we describe a new

[1] Recall that, in an execution based formalization, one views a memory as a machine generating strings over the alphabet of completed instructions.

formulation of the interleaved-sequentiality checking and prove it to be equivalent to the original. In Section 4.5, by making use of the result of the preceding section, we propose a way of pruning the *irrelevant* instructions of a noninterleaved-sequential execution. In Section 4.6, we prove that only a finite number of executions has to be checked to conclude the nonexistence of any unambiguous noninterleaved-sequential execution. We conclude the chapter with a summary of the results.

4.2 Notation

A shared memory implementation (smi, for short) is a system parameterized over the sets of processors, addresses and data values; P, A and D, respectively. Since we are only interested in implementations, we will take P, A and D as finite subsets of $\mathbb{N}$, with the further assumption, for convenience, that $[|P|] = P$ (same for A and D). When we talk about a single smi, the parameter values are understood to be some P, A and D.

In this chapter, for the first part, we need not be concerned with the input part of an execution, unlike the previous chapter. We will abstract away the input and formulate the problem of checking interleaved-sequentiality in terms of executions.

Let S be an smi. Its alphabet, Σ_S, consists of two classes of symbols. The first class, R_S, consists of read events. These are the symbols of the form $\texttt{r(}p\texttt{,}a\texttt{,}d\texttt{)}$[2] where $p \in P$ is the (index of) the processor that owns the read, $a \in A$ is the address that is being queried and $d \in D \cup \{0\}$ is the value returned for this read. The second class, W_S, consists of write events. These are the symbols of the form $\texttt{w(}p\texttt{,}a\texttt{,}d\texttt{)}$[3], where p and a are as above and $d \in D$[4]. Hence, Σ_S, is the union of R_S and W_S. We will usually drop the subscripts when no confusion is likely to arise. We will use the variables p, p_1, p_2, etc. ranging over P, a, b, a_1, a_2, etc. ranging over A, d, d', d_1, d_2, etc. over D, i, j, etc. over $\mathbb{N}$.

[2] This is equivalent to $\texttt{(r}_\circ\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}$ of the previous chapters. We are using this new notation to emphasize that we are abstracting away input.

[3] Same as $\texttt{(w}_\circ\texttt{,}p\texttt{,}a\texttt{,}d\texttt{)}$.

[4] Without loss of generality, we are assuming that the value $\texttt{0}$ is reserved for the initial value of each address.

We will also define other partitions on a given smi alphabet Σ. Let $\Sigma^P_{p_j}$ be the set of all symbols of the form $\mathtt{r}(p_j, a, d)$ or $\mathtt{w}(p_j, a, d)$ for $a \in A$, $d \in D$. This is the partition based on processor, and each $\Sigma^P_{p_j}$ has only and all of events (read and write alike) that belong to processor p_j. Similarly, define $\Sigma^A_{a_j}$ to be the set of all events that are to the address a_j; $\Sigma^D_{d_j}$ to be the set of all events that have the data value d_j.

For any binary relation R over a set Z, we define the directed graph $G_R = (V_R, E_R)$ such that $V_R = Z$ and $E_R = R$. In such a case, graph G_R is said to represent the relation R.

For a given smi, a sequential execution of a processor p_i, $p_i \in P$, is a labelled, directed (acyclic) graph, $G_{p_i} = (V_{p_i}, E_{p_i}, \lambda_{p_i})$, where λ_{p_i} is a mapping from V_{p_i} to $\Sigma^P_{p_i}$. As is custom, V_{p_i} is the set of vertices of G_{p_i} and $E_{p_i} \subset V^2_{p_i}$, the set of edges. We require that (V_{p_i}, E_{p_i}) be a graph representing a total order over V_{p_i}.

Given two labelled graphs, G_1, G_2, which have disjoint vertex and edge sets, by their union, we mean the labelled graph $G_u = (V_u, E_u, \lambda_u)$, where $V_u = V_{G_1} \cup V_{G_2}$, $E_u = E_{G_1} \cup E_{G_2}$ and $\lambda_u(v)$ is equal to λ_{G_1} if $v \in V_{G_1}$, equal to λ_{G_2}, otherwise.

A concurrent execution of an smi S, $G_c = (V_c, E_c, \lambda_c)$, is the union of a collection of sequential executions for each p_i[5], $p_i \in P$. We will call the concurrent execution *unambiguous* if for any $v_1, v_2 \in V_c$ such that $\lambda_c(v_1), \lambda_c(v_2) \in W \cap \Sigma^A_a$ for some $a \in A$, we have $\lambda_c(v_1) \in \Sigma^D_d$ imply $\lambda_c(v_2) \notin \Sigma^D_d$, for all $d \in D$. That is, a concurrent execution is unambiguous if no two writes have the same address and data values.

A concurrent execution $G = (V, E, \lambda)$ is *legal* if for any $v \in V$ such that $\lambda(v) \in R$, either $\lambda(v) \in \Sigma^D_0$ or $\lambda(v) \in \Sigma^D_d \cap \Sigma^A_a$ and there exists $v' \in V$ with $\lambda(v') \in \Sigma^D_d \cap \Sigma^A_a \cap W$.

We shall be dealing only with legal and unambiguous concurrent executions in the rest of this chapter.

We will usually employ diagrams to represent concurrent executions instead of an extensional (and clumsy) description of the sets V, E and the labelling function

[5] We note that each pair of graphs has mutually disjoint vertex and edge sets.

λ. An explanation of some conventions used in these diagrams is in order. A typical diagram is given in Fig. 4.1. We will let $V \subset \mathbb{N}$. Each vertex is represented by its value inside a circle. Each sequential execution is arranged vertically into a column which is annotated by the name of the processor to which it belongs. For instance, the vertices `1`, `2`, `3` all belong to the sequential execution of processor 1, denoted by P_1, `4` to processor 2, P_2, etc. To make the diagrams more readable, we do not draw all the edges; the total order (per sequential execution) would be the transitive closure of the drawn edges. The label of each vertex is written next to it. In Fig. 4.1, the labels of vertices `2` and `5` are `r(1,`b`,0)` and `w(3,`c`,2)`, respectively. The labels will also be referred to as *instructions* or *responses* depending on the context.

4.3 Interleaved-Sequentiality Checking: Present Situation

In this section, we will describe *interleaved-sequentiality* of a concurrent execution using the notation presented in this chapter. Let us recall that the original definition, given by Lamport in [43], states that an execution is interleaved-sequential if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." In what follows, we will formally state this definition.

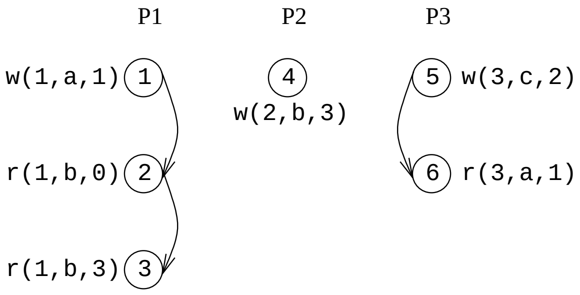


Figure 4.1. Sample concurrent execution, G_0.

Definition 4.1 (interleaved-sequential) *Let $G = (V, E, \lambda)$ be an (unambiguous) concurrent execution. It is called interleaved-sequential if there exists a total order $<_{sc}$ over V such that*

1. *For any $v \in V$ such that $\lambda(v) \in R \cap \Sigma_a^A$ for some $a \in A$, one of the following holds*

   (a) $\lambda(v) \in \Sigma_0^D$ and there does not exist v' such that $\lambda(v') \in W \cap \Sigma_a^A$ and $v' <_{sc} v$.

   (b) $\lambda(v) \in \Sigma_d^D$, $d \neq 0$, there is a $v_1 \in V$ with $\lambda(v_1) \in W \cap \Sigma_a^A \cap \Sigma_d^D$ and $v_1 <_{sc} v$, and there does not exist $v_2 \in V$ such that $\lambda(v_2) \in \Sigma_a^A \cap \Sigma_{d'}^D$, $d \neq d'$ and $v_1 <_{sc} v_2 <_{sc} v$[6]*.*

2. *$(v_1, v_2) \in E$ implies $v_1 <_{sc} v_2$.*

Instead of generating a total order on all the vertices of a concurrent execution, the existence of a certain partial order is enough to guarantee interleaved-sequentiality. Let G be a concurrent execution. Let W_a be the set of vertices v such that $\lambda(v) \in \Sigma_a^A \cap W$. For any $v \in V$, we let $\omega(v)$ be v if $\lambda(v) \in W \cup \Sigma_0^D$. If $\lambda(v) = \mathtt{r(}p_i\mathtt{,}a\mathtt{,}d\mathtt{)}$ and $d \neq 0$, then $\omega(v) = v'$, where $\lambda(v') = \mathtt{w(}p_j\mathtt{,}a\mathtt{,}d\mathtt{)}$, for some $p_j \in P$ and $v' \in V$; that is, for $\lambda(v) \in \Sigma_d^D \cap \Sigma_a^A$, $\omega(v)$ gives the vertex whose label is in $W \cap \Sigma_d^D \cap \Sigma_a^A$.[7] Let T_a be an arbitrary total order on W_a, for each $a \in A$. Let I_a be a relation over V^2 such that $(v_1, v_2) \in I_a$ if one of the following holds:

- $(v_1, v_2) \in T_a$.
- $\lambda(v_1) \in W$, $\lambda(v_2) \in R$ and $v_1 = \omega(v_2)$.
- There is a $v_3 \in T_a$ such that $v_3 = \omega(v_1)$ and $(v_3, v_2) \in T_a$.

[6] The existence of such a v_1 implies that G is legal.

[7] By virtue of unambiguity, this mapping is well-defined.

Let IC_a be the transitive closure of I_a. Then, the *coh-augmented* concurrent execution (coce) of a concurrent execution G is the graph $G_t = (V, E_t, \lambda)$ where $(v_1, v_2) \in E_t$ if and only if either $(v_1, v_2) \in E$ or $(v_1, v_2) \in IC_a$ for some $a \in A$.

It was shown in [53] that a concurrent execution is interleaved-sequential if and only if its coce is acyclic.[8] Unfortunately, this result does not bear much on the formal verification of sequential consistency. There is not a single graph to be checked as evidenced by the plurality of the candidates for the T_a relation defined above.

Let $iter_G(Rel)$ for a relation Rel and a concurrent execution G be defined such that $(v_1, v_2) \in iter_G(Rel)$, for $v_1, v_2 \in V$, if one of the following holds:

1. $(v_1, v_2) \in Rel$
2. $v_1 = \omega(v_2)$ and $v_2 \neq \omega(v_2)$.
3. There exists $v_3 \in V$ such that $(v_3, v_2) \in Rel$ and $\lambda(v_1), \lambda(v_3) \in \Sigma_a^A \cap \Sigma_d^D$, for some $a \in A$, $d \in D$.

Let I_{init} over V^2 be defined such that $(v_1, v_2) \in I_{init}$ if and only if $\lambda(v_1) \in \Sigma_0^D \cap \Sigma_a^A$ and $\lambda(v_2) \in \Sigma_a^A \cap \Sigma_d^D$, for some $a \in A$ and $0 \neq d \in D$. Then, let IR be the transitive closure of the fix-point of $iter_G(E \cup I_{init})$.

The work in [50] was aimed to improve the result of [53]. It was claimed that IR as defined above would be enough; that is, the graph representing IR itself would be acyclic if and only if the concurrent execution is interleaved-sequential. Unfortunately, that result turned out to be erroneous[33].

As an illustration of the above ideas, we provide a sample concurrent execution, G_f, which is a simplified version of a counter example to the method of [50]. The concurrent execution is given in Fig. 4.2.

In Fig. 4.3, the graph representing the relation IR is given. The edges due to E_f are solid, whereas the other edges are depicted by the dashed edges. All

[8] Actually, in [57], a concurrent execution is *defined* to be interleaved-sequential, if its coce is acyclic.

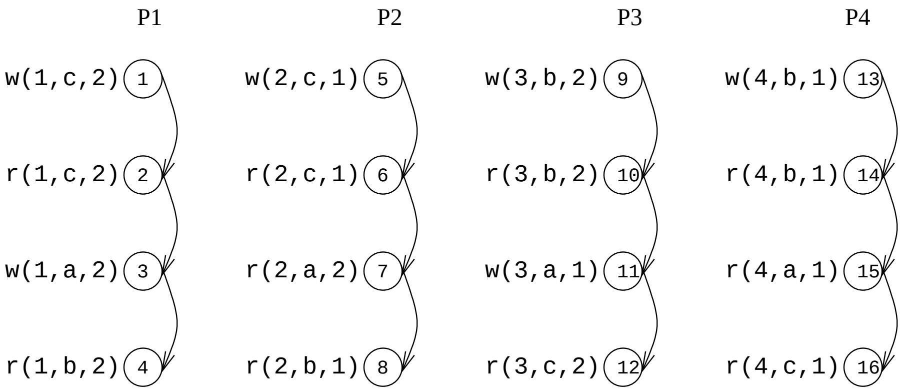


Figure 4.2. The concurrent execution G_f.

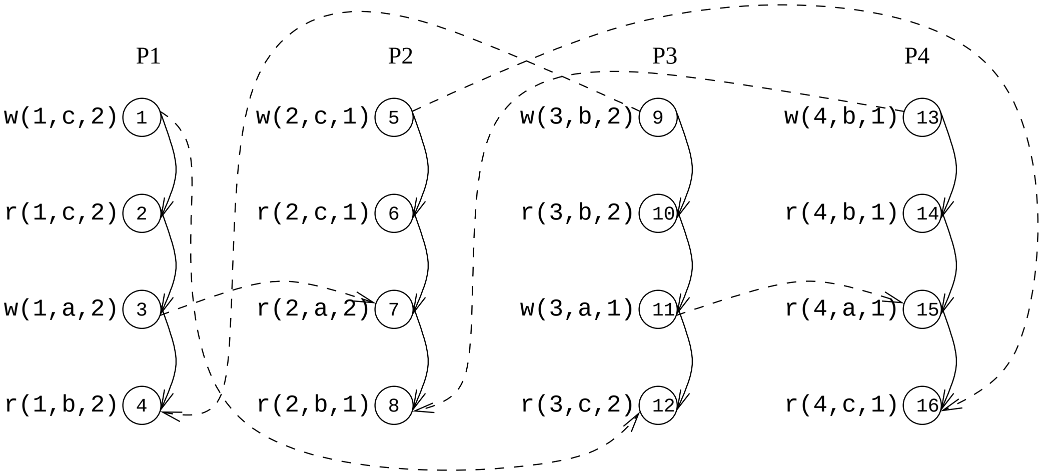


Figure 4.3. The graph representing the relation IR of [33].

of these dashed edges are due to the second item of the definition of IR. For instance, there is an edge from 1 to 12, as $1 = \omega(12)$. As can be seen from this figure, the graph is acyclic, even though as we will see, it does not correspond to an interleaved-sequential concurrent execution.

In Fig. 4.4, we show part of the graph which corresponds to the coce of G_f when T_a is $\{(3, 11)\}$ (for address a, the write of 2 is ordered before the write of 1), T_b is $\{(13, 9)\}$ (for address b, the write of 1 is ordered before the write of 2),

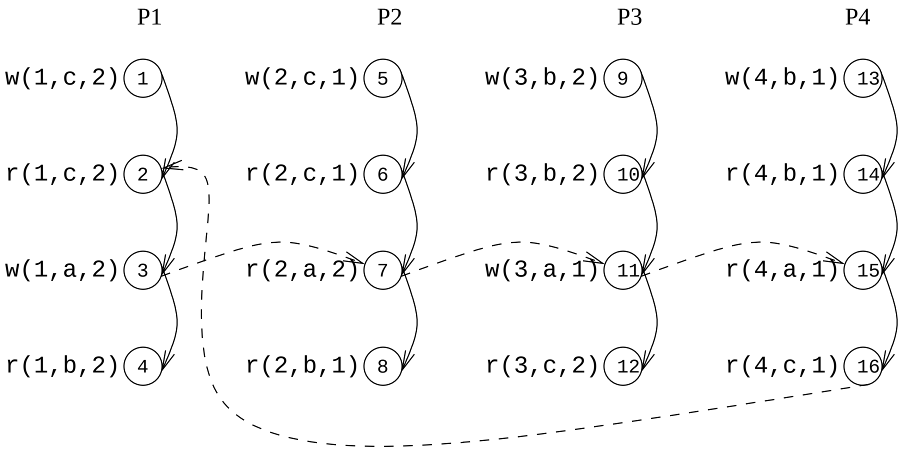


Figure 4.4. Part of the coce of G_f exhibiting a cycle, for $T_a = \{(\texttt{3}, \texttt{11})\}$, $T_b = \{(\texttt{13}, \texttt{9})\}$, $T_c = \{(\texttt{5}, \texttt{1})\}$.

T_c is $\{(\texttt{5}, \texttt{1})\}$ (for address c, the write of 1 is ordered before the write of 2). The loop given by 2, 3, 7, 11, 15, 16, 2 shows that the orderings given by T_a, T_b and T_c cannot be extended to a total order satisfying the requirements of the definition of interleaved-sequentiality.

4.4 Constraints for Interleaved-Sequentiality: A New Formulation

In this section, we will define a new way to check the interleaved-sequentiality of a concurrent execution. The idea is to transform a concurrent execution to a set of (impossibility) constraints, which we call the *constraint satisfaction problem*, csp for short. We will prove that a concurrent execution is interleaved-sequential if and only if its csp is satisfiable.

Before getting into the specifics, an intuitive explanation is in order. We said that a concurrent execution is a combination of sequential executions, one per processor[9] 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,

[9] No emphasis should be placed on the word "processor"; it is just a name. We could be using *thread* or *process* in its place.

is a fictitious order that conforms to all the requirements enforced by each processor. But what exactly do we mean by these requirements?

Look at the concurrent execution G_1 of Fig. 4.5. 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: $\texttt{w(}a\texttt{,1)}$ (shorthand for $\texttt{w(}p\texttt{,}a\texttt{,1)}$ for some $p \in P$; since we are dealing with unambiguous runs exclusively, there is at most one such write) must precede $\texttt{w(}a\texttt{,2)}$. 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.

Now, look at a snippet of a concurrent execution G_2, in Fig. 4.6. One requirement is that $\texttt{r(1,}a\texttt{,1)}$ precede $\texttt{r(1,}b\texttt{,1)}$. By legality, it is also required that $\texttt{w(}a\texttt{,1)}$ and $\texttt{w(}b\texttt{,1)}$ exist. However, it does not seem to relate these writes. We might conclude that these are the only the requirements 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 as *forbidden* orderings. For instance, in Fig. 4.5, we could say that processor 1 forbids the ordering where $\texttt{w(}a\texttt{,2)}$ precedes $\texttt{w(}a\texttt{,1)}$. In the case of binary orderings, the difference is superfluous. However, for Fig. 4.6, if we say that, for any other write to b, $\texttt{w(}b\texttt{,}d\texttt{)}$ such that $d \neq 1$, we cannot have $\texttt{w(}b\texttt{,1)}$ precede $\texttt{w(}b\texttt{,}d\texttt{)}$ when both precede $\texttt{w(}a\texttt{,1)}$, we introduce a new requirement.

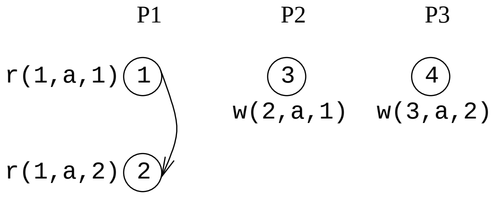


Figure 4.5. Sample concurrent execution, G_1.

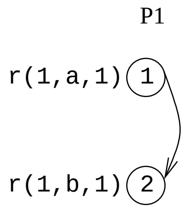


Figure 4.6. Sample concurrent execution, G_2.

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.

Definition 4.2 *Let $G = (V, E, \lambda)$ be a concurrent execution. Let v_1, v_2 be in V such that $(v_1, v_2) \in E$. Let $i_j = \lambda(v_j)$, for $j = 1, 2$. The constraint set for (v_1, v_2), $\kappa((v_1, v_2))$, is:*

1. *If i_2 is $\mathtt{r}(p,b,d_2)$ and*

   (a) *If i_1 is either $\mathtt{w}(p,a,d_1)$ or $\mathtt{r}(p,a,d_1)$, and $a \neq b$, then*

   $$\begin{aligned}\{&\mathtt{w}(b,d_2) \prec \mathtt{w}(b,d_3) \prec \mathtt{w}(a,d_1) \mid \\ &\exists p' \in P, v' \in V, d_3 \in D\ . \\ &\lambda(v') = \mathtt{w}(p',b,d_3) \wedge d_3 \neq d_2\}\end{aligned}$$

   (b) *If i_1 is either $\mathtt{w}(p,b,d_2)$ or $\mathtt{r}(p,b,d_2)$, then*

   $$\emptyset$$

   (c) *If i_1 is either $\mathtt{w}(p,b,d_1)$ or $\mathtt{r}(p,b,d_1)$, and $d_1 \neq d_2$, then*

   $$\{\mathtt{w}(b,d_2) \prec \mathtt{w}(b,d_1)\}$$

2. *If i_2 is $\mathtt{w}(p,b,d_2)$ and*

   (a) *If i_1 is either $\mathtt{w}(p,a,d_1)$ or $\mathtt{r}(p,a,d_1)$, and $a \neq b$, then*

   $$\{\mathtt{w}(b,d_2) \prec \mathtt{w}(a,d_1)\}$$

(b) If i_1 *is* $\texttt{r(}p\texttt{,}b\texttt{,}d_2\texttt{)}$*, then*

$$\{\texttt{w(}b\texttt{,}d_2\texttt{)} \prec \texttt{w(}b\texttt{,}d_2\texttt{)}\}$$

(c) If i_1 *is either* $\texttt{w(}p\texttt{,}b\texttt{,}d_1\texttt{)}$ *or* $\texttt{r(}p\texttt{,}b\texttt{,}d_1\texttt{)}$*, and* $d_1 \neq d_2$*, then*

$$\{\texttt{w(}b\texttt{,}d_2\texttt{)} \prec \texttt{w(}b\texttt{,}d_1\texttt{)}\}$$

For any $v \in V$*, let* $\iota(v) = \{\texttt{w(}a, d_1\texttt{)}\prec\ \texttt{w(}a\texttt{,0)}\}$*, when* $\lambda(v)$ *is either* $\texttt{w(}p\texttt{,}a\texttt{,}d_1\texttt{)}$ *or* $\texttt{r(}p\texttt{,}a\texttt{,}d_1\texttt{)}$*,* $d_1 \neq 0$*; otherwise,* $\iota(v) = \emptyset$*.*

In the above definition, the expressions $x \prec y$ or $x \prec y \prec z$ are called *constraint*s. The terms x,y,z (x,y) are said to *appear* in the constraint $x \prec y \prec z$ ($x \prec y$). We say that x appears in a constraint set if the set contains a constraint in which x appears. Let the elements of a constraint set C, **elem**(C), be the set of all terms which appear in C.

For a concurrent execution G_c, we define its constraint set, $\mathcal{CS}_c$, as the union of the sets $\bigcup_{e\in E_c} \kappa(e)$ and $\bigcup_{v\in V_c} \iota(v)$.

Example 3 *Let us consider the constraints for the concurrent execution* G_f *of Fig. 4.2. In the sequential execution of processor 1,* $P1$*, there are 4 responses: the vertices* $\texttt{1}$*,* $\texttt{2}$*,* $\texttt{3}$*,* $\texttt{4}$*. Some of the constraints due to* κ *for* $P1$*, then, are given as follows:*

1. $\kappa((\texttt{1},\texttt{2})) = \emptyset$*. The rule for this edge is given by 1(b) in the definition of* κ*.*

2. $\kappa((\texttt{1},\texttt{3})) = \{\texttt{w(}a\texttt{,2)} \prec \texttt{w(}c\texttt{,2)}\}$*. The rule for this edge is given by 2(a).*

3. $\kappa((\texttt{1},\texttt{4})) = \{\texttt{w(}b\texttt{,2)} \prec \texttt{w(}b\texttt{,1)} \prec \texttt{w(}c\texttt{,2)}\}$*. The rule for this edge is given by 1(a). Note that, since there are only two distinct values,* $\texttt{1}$ *and* $\texttt{2}$*, written to address* b*, the constraint for this edge is a singleton.*

The remaining constraints for G_f *due to* κ *are formed similarly.*

As for ι*, we have* $\iota(\texttt{1}) = \iota(\texttt{2}) = \iota(\texttt{12}) = \{\texttt{w(}c\texttt{,2)} \prec \texttt{w(}c\texttt{,0)}\}$*. Note that, for* $v, v' \in V_f$*, if* $\omega(v) = \omega(v')$*, then* $\iota(v) = \iota(v')$*. For* G_f*, there are six different sets formed by* ι*, as there are six different writes.*

Definition 4.3 (Constraint satisfaction) *A set of constraints, C, is satisfiable if there exists a total order $<_t$ over* $\mathbf{elem}(C)$ *such that*

1. *If $t_1 \prec t_2 \in C$, then $t_2 <_t t_1$.*

2. *If $t_1 \prec t_2 \prec t_3 \in C$, then either $t_2 <_t t_1$ or $t_3 <_t t_2$.*

In such a case, $<_t$ is said to satisfy C.

Example 4 *Let us consider G_f. We have the following constraints in the set CS_f for G_f:*

$$\left.\begin{array}{l}
\mathtt{w}(a,2) \prec \mathtt{w}(c,2), \\
\mathtt{w}(b,2) \prec \mathtt{w}(b,1) \prec \mathtt{w}(c,2), \\
\mathtt{w}(b,2) \prec \mathtt{w}(b,1) \prec \mathtt{w}(a,2),
\end{array}\right\} \quad P1$$
$$\left.\begin{array}{l}
\mathtt{w}(a,2) \prec \mathtt{w}(a,1) \prec \mathtt{w}(c,1), \\
\mathtt{w}(b,1) \prec \mathtt{w}(b,2) \prec \mathtt{w}(c,1), \\
\mathtt{w}(b,1) \prec \mathtt{w}(b,2) \prec \mathtt{w}(a,2),
\end{array}\right\} \quad P2$$
$$\left.\begin{array}{l}
\mathtt{w}(a,1) \prec \mathtt{w}(b,2), \\
\mathtt{w}(c,2) \prec \mathtt{w}(c,1) \prec \mathtt{w}(b,2), \\
\mathtt{w}(c,2) \prec \mathtt{w}(c,1) \prec \mathtt{w}(a,1),
\end{array}\right\} \quad P3$$
$$\left.\begin{array}{l}
\mathtt{w}(a,1) \prec \mathtt{w}(a,2) \prec \mathtt{w}(b,1), \\
\mathtt{w}(c,1) \prec \mathtt{w}(c,2) \prec \mathtt{w}(b,1), \\
\mathtt{w}(c,1) \prec \mathtt{w}(c,2) \prec \mathtt{w}(a,1)
\end{array}\right\} \quad P4$$

The first three constraints are due to the responses in processor $P1$, the second three due to $P2$, etc. Let $<_t$ be given by:

$$\mathtt{w}(c,2) <_t \mathtt{w}(a,2) <_t \mathtt{w}(b,1) <_t \mathtt{w}(b,2) <_t \mathtt{w}(a,1) <_t \mathtt{w}(c,1)$$

Then, the first constraint $\mathtt{w}(a,2) \prec \mathtt{w}(c,2)$ is satisfied because $\mathtt{w}(c,2) <_t \mathtt{w}(a,2)$. The second constraint $\mathtt{w}(b,2) \prec \mathtt{w}(b,1) \prec \mathtt{w}(c,2)$ is satisfied because $\mathtt{w}(c,2) <_t \mathtt{w}(b,2)$. Continuing in this manner, we see that all the constraints due to the responses of $P1$, $P3$ and $P4$ are satisfied. However, the first constraint due to $P2$, $\mathtt{w}(a,2) \prec \mathtt{w}(a,1) \prec \mathtt{w}(c,1)$ is not satisfied as we also have $\mathtt{w}(a,2) <_t \mathtt{w}(a,1) <_t \mathtt{w}(c,1)$. Since, none of the 6! possible orderings of the writess forms a satisfying total order, we conclude that CS_f is not satisfiable. It is no coincidence that G_f is not i-s, as the next theorem demonstrates.

Let $\omega_c(v) = \texttt{w(}a\texttt{,}d\texttt{)}$ if and only if $\omega(v) = \texttt{w(}p\texttt{,}a\texttt{,}d\texttt{)}$ for some $p \in P$. Let $WC_c = \{\texttt{w(}a\texttt{,}d\texttt{)} \mid \texttt{w(}a\texttt{,}d\texttt{)} \in \mathbf{elem}(\mathcal{CS}_c), d \neq 0\}$. The set WC_c is isomorphic to $\mathbf{elem}(\mathcal{CS}_c)$ with the (fictitious) initial writes removed. Let $\omega_p(\texttt{w(}p\texttt{,}a\texttt{,}d\texttt{)}) = \texttt{w(}a\texttt{,}d\texttt{)}$. Note that ω_p is a bijection from $\lambda_c(V_c) \cap W$ to WC_c. We are now ready to state the main theorem of this chapter.

Theorem 4.1 *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.*

Proof (Theorem 4.1): $(\Rightarrow)$: Let $<_{sc}$ be the logical order of G_c. First, augment $<_{sc}$ so that it is defined for all elements of $\mathbf{elem}(\mathcal{CS}_c)$. Let $<_o$ be an arbitrary total order over $\{\texttt{w(}a\texttt{,0)} \mid a \in A\}$. Let $<_c$ be the order defined by

1. Let $v_1, v_2 \in V_c$ be such that $\lambda_c(v_1), \lambda_c(v_2) \in W$. If $v_1 <_{sc} v_2$, then $\omega_p(\lambda_c(v_1)) <_c \omega_p(\lambda_c(v_2))$.

2. if $t_1 <_o t_2$, then $t_1 <_c t_2$.

3. if $t_1 = \texttt{w(}a\texttt{,0)}$, $t_2 = \texttt{w(}b\texttt{,}d\texttt{)}$ and $d \neq 0$, then $t_1 <_c t_2$.

It is clear that $<_c$ is also a total order. It can be proven that $<_c$ satisfies $\mathcal{CS}_c$ by contradiction. Assume it does not; go over all possible constraints, using the definition of the constraint set and interleaved-sequentiality, and show that in each case, unsatisfiability implies a violation of interleaved-sequentiality. To illustrate, let us assume that $\texttt{w(}a_1\texttt{,}d_1\texttt{)}\prec\texttt{w(}a_1\texttt{,}d_2\texttt{)}\prec\texttt{w(}a_2\texttt{,}d_3\texttt{)}\in \mathcal{CS}_c$ is a constraint that is not satisfied by $<_c$, that is $\texttt{w(}a_1\texttt{,}d_1\texttt{)}<_c\texttt{w(}a_1\texttt{,}d_2\texttt{)}<_c\texttt{w(}a_2\texttt{,}d_3\texttt{)}$. Let us assume that the constraint was generated by $(v_1, v_2) \in E_c$ such that $\lambda_c(v_1) = \texttt{r(}p_1\texttt{,}a_2\texttt{,}d_3\texttt{)}$, $\lambda_c(v_2) = \texttt{r(}p_1\texttt{,}a_1\texttt{,}d_1\texttt{)}$ and $a_1 \neq a_2$. Let $v_3, v_4, v_4 \in V_c$ such that $v_3 = \omega(v_1)$, $v_5 = \omega(v_2)$, $\lambda_c(v_4) = \omega_p^{-1}(\texttt{w(}a_1\texttt{,}d_2\texttt{)})$.[10]. By int-seq(3), we have $v_1 <_{sc} v_2$. There are 4 cases to consider:

1. If $d_1 = d_3 = 0$, $\texttt{w(}a_2\texttt{,}d_3\texttt{)}<_c\texttt{w(}a_1\texttt{,}d_2\texttt{)}$, for any $d_2 \neq 0$, by the definition of $<_c$. So, this case is not possible.

[10] By the definition of κ (item 1(a)), v_3, v_4, v_5 exist.

2. If $d_1 \neq 0$ and $d_3 = 0$, then $\mathtt{w(}a_2\mathtt{,}d_3\mathtt{)}<_c\mathtt{w(}a_1\mathtt{,}d_1\mathtt{)}$, again by the definition of $<_c$. This case is not possible.

3. If $d_1 = 0$ and $d_3 \neq 0$, then

   (a) $v_3 <_{sc} v_1 <_{sc} v_2$, by the int-seq(2-b,3).

   (b) $v_2 <_{sc} v_4$, by int-seq(2-a).

   (c) $v_3 <_{sc} v_2 <_{sc} v_4$, by (a) and (b).

   So we have $\mathtt{w(}a_2\mathtt{,}d_3\mathtt{)}<_c\mathtt{w(}a_1\mathtt{,}d_2\mathtt{)}$, which contradicts the assumption.

4. If $d_1, d_3 \neq 0$, then

   (a) $v_5 <_{sc} v_2$, by int-seq(2-b).

   (b) $v_3 <_{sc} v_1$, by int-seq(2-b).

   (c) $v_5 <_{sc} v_2 <_{sc} v_4$, by $\mathtt{w(}a_1\mathtt{,}d_1\mathtt{)}<_c\mathtt{w(}a_1\mathtt{,}d_2\mathtt{)}$ and int-seq(2-b).

   (d) $v_1 <_{sc} v_2$, by assumption.

   Combining (a-d) above, we get $v_3 <_{sc} v_2 <_{sc} v_4$, which again results in a contradiction.

The remaining cases are proved similarly.

($\Leftarrow$): Let $\mathcal{CS}_c$ be satisfiable. Then, by definition, there is a total order over $\mathbf{elem}(\mathcal{CS}_c)$ that satisfies $\mathcal{CS}_c$. Let $<_{t'}$ be that total order. Define $<_t$ to be the total order over $\lambda_c(V_c) \cap W$ such that if $w_1 = \mathtt{w(}a_1\mathtt{,}d_1\mathtt{)}$, $w_2 = \mathtt{w(}a_2\mathtt{,}d_2\mathtt{)}$, $w_1' = \mathtt{w(}p_1\mathtt{,}a_1\mathtt{,}d_1\mathtt{)} = \omega_p^{-1}(w_1)$, $w_2' = \mathtt{w(}p_2\mathtt{,}a_2\mathtt{,}d_2\mathtt{)} = \omega_p^{-1}(w_2)$ and $w_1 <_{t'} w_2$, then $w_1' <_t w_2'$. Note that, by the unambiguity of the concurrent execution, the order $<_t$ is well-defined. The set WC is isomorphic to $\mathbf{elem}(\mathcal{CS}_c)$ with the (fictitious) initial value writes removed. Using $<_t$, define a partial order $<_p$ over V_c as follows:

1. If $\lambda_c(v_1) <_t \lambda_c(v_2)$, then $v_1 <_p v_2$.

2. Let $\lambda_c(v_1)=\mathtt{w(}p_1\mathtt{,}a\mathtt{,}d_1\mathtt{)}$, $\lambda_c(v_2)=\mathtt{w(}p_2\mathtt{,}a\mathtt{,}d_2\mathtt{)}$ and $\lambda_c(v_1) <_t \lambda_c(v_2)$. Then, for any $v_3 \in V_c$ with $\lambda_c(v_3) \in R$ and $\omega(v_3) = v_1$, we have both $v_1 <_p v_3$ and $v_3 <_p v_2$.

3. If we have both $\lambda_c(v_1) = \texttt{r(}p_1\texttt{,}a\texttt{,0)}$ and $\lambda_c(v_2) = \texttt{w(}p_2\texttt{,}a\texttt{,}d\texttt{)}$, then $v_1 <_p v_2$ holds.

We will refer to the first requirement as def-p(1), to the second as def-p(2), etc. Observe that, for two vertices v_1 and v_2:

1. if $\lambda_c(v_1)$ and $\lambda_c(v_2)$ are both writes, $v_1 <_p v_2$ if and only if $\lambda_c(v_1) <_t \lambda_c(v_2)$.
2. if $\lambda_c(v_1)$ and $\lambda_c(v_2)$ are both reads, then they are not ordered by $<_p$.
3. If $\lambda_c(v_1)$ is a read and $\lambda_c(v_2)$ is a write, then they are ordered by $<_p$ if and only if both are to the same address.

We will refer to the first observation as obs(1), to the second as obs(2), etc.

Lemma 4.1 *There does not exist a set of vertices, $v_i \in V_c$, such that $v_1 <_p v_2 <_p \ldots <_p v_n <_p v_1$, for any $n \in |V_c|$.*

Proof (Lemma 4.1): We will prove by induction on n.

Base case ($n = 1$): Then, we must have $v_1 <_p v_1$. This cannot be the result of the application of (1) in the definition of $<_p$ as that would contradict the fact that $<_t$ is a total order. The remaining cases order different vertices; hence, cannot be the reason for this ordering. Therefore, n cannot be 1.

Induction hypothesis: For all sets of cardinality k or less, assume the claim holds.

Induction step ($n = k + 1$): Then, we have $v_1 <_p \ldots <_p v_{k+1} <_p v_1$. Note that, both $\lambda_c(v_1)$ and $\lambda_c(v_{k+1})$ cannot be both reads by obs(2). Let us examine the possibilities for $r_1 = \lambda_c(v_1)$ and $r_{k+1} = \lambda_c(v_{k+1})$:

1. Assume that $r_1 = \texttt{r(}p_1\texttt{,}a_1\texttt{,}d_1\texttt{)}$, $r_{k+1} = \texttt{w(}p_2\texttt{,}a_1\texttt{,}d_2\texttt{)}$. Then, since $v_{k+1} <_p v_1$, they must be ordered due to def-p(2) and $d_1 = d_2$. This implies that $\lambda_c(v_2)$ must be a write (by obs(2)). Due to the premise of def-p(2), we must have $r_{k+1} <_t \lambda_c(v_2)$, which would imply $v_{k+1} <_p v_2$. But, then $v_2 <_p v_3 <_p \ldots <_p v_{k+1} <_p v_2$ with only k vertices holds. By induction hypothesis, this cannot be.

2. Assume that $r_1 = \mathtt{w(}p_1\mathtt{,}a_1\mathtt{,}d_1\mathtt{)}$, $r_{k+1} = \mathtt{r(}p_2\mathtt{,}a_1\mathtt{,}d_2\mathtt{)}$. There are two cases to consider:

   (a) $d_2 = 0$. Examining def-p(1-3), we see that we cannot have another vertex ordered before v_{k+1}; in particular, $v_k <_p v_{k+1}$ cannot hold.

   (b) $d_2 \neq 0$. Then, by obs(2,3), it is easy to see that $v_k = \omega(v_{k+1})$. By def-p(1), we have $r_k <_p r_1$. Hence, $v_1 <_p v_2 <_p \ldots <_p v_k <_p v_1$ with k vertices should hold. By induction hypothesis, this cannot be.

   So, this case is not possible.

3. Assume that $r_1 = \mathtt{w(}p_1\mathtt{,}a_1\mathtt{,}d_1\mathtt{)}$, $r_{k+1} = \mathtt{w(}p_2\mathtt{,}a_2\mathtt{,}d_2\mathtt{)}$. There are two cases to consider:

   (a) $r_2 = \lambda_c(v_2) = \mathtt{w(}p_3\mathtt{,}a_3\mathtt{,}d_3\mathtt{)}$, which would mean that $v_1 <_p v_2$ due to def-p(1), because of obs(1). Again, by obs(1), we have $r_{k+1} <_t r_1 <_t r_2$, which implies that $r_{k+1} <_t r_2$ by the totality of $<_t$. This in turn implies that $v_{k+1} <_p v_2$. Then, $v_2 <_p \ldots <_p v_{k+1} <_p v_2$ with k vertices holds. By induction hypothesis, this cannot be.

   (b) $r_2 = \lambda_c(v_2) = \mathtt{r(}p_3\mathtt{,}a_3\mathtt{,}d_3\mathtt{)}$. Then, by obs(3), $a_1 = a_3$. Also, by def-p(2), we have $d_1 = d_3$. Then, $r_3 = \lambda_c(v_3) = \mathtt{w(}p_4\mathtt{,}a_1\mathtt{,}d_4\mathtt{)}$, by obs(2-3). But, this means that $r_1 <_t r_3$, by def-p(2). We also have, by assumption, $r_{k+1} <_t r_1$. By the transitivity of $<_t$, we have $r_{k+1} <_t r_3$, which implies $v_{k+1} <_p v_3$, by def-p(1). Then, $v_3 <_p \ldots <_p v_{k+1} <_p v_3$ with $k-1$ vertices, holds. By induction hypothesis, this cannot be.

4. The other cases where r_1 and r_{k+1} are to different addresses and at least one of them is a read, cannot occur as by obs(3), they are not ordered by $<_p$.

This completes the induction step as all possible cases are covered.

□

Let $<_p^+$ denote the transitive closure of $<_p$. It follows from Lemma 4.1 that $<_p^+$ is a strict partial order; that is, it is irreflexive, antisymmetric and transitive.

Now, if two vertices are not ordered by $<_p^+$, then either they are accessing different addresses and at least one of them is a read, or they are both reads and have the same address and data values. So, the next step is to augment this partial order yet to another partial order, so that the sequential order per processor is included.

Let $<_s$ be a relation over V_c satisfying the following:

1. If $v_1 <_p^+ v_2$, then $v_1 <_s v_2$.
2. If $(v_1, v_2) \in E_c$, then $v_1 <_s v_2$.

We refer to the first requirement as def-s(1), to the second as def-s(2).

This way, $<_s$ augments $<_p^+$ with the sequential order as mentioned above. Similar to $<_p^+$, it can be shown the the transitive closure of $<_s$, $<_s^+$, is also a strict partial order.

Lemma 4.2 *There does not exist a set of vertices $v_i \in V_c$, such that $v_1 <_s v_2 <_s \ldots v_n <_s v_1$, for any $n \in |V_c|$.*

Proof (Lemma 4.2): We prove it by induction on n.

Base case ($n = 1$): $v_1 <_s v_1$. This cannot be due to def-s(1), as $<_p^+$ was proved to be irreflexive. It cannot be due to def-s(2) either as E_c is acyclic. So, such an ordering does not exist.

Induction hypothesis: Assume that the claim holds for all n less than or equal to k.

Induction step ($n = k + 1$): Then, we have $v_1 <_s v_2 <_s \ldots <_s v_{k+1} <_s v_1$. Note that, if $v_i <_p^+ v_{i+1}$, then $v_{i+1} <_p^+ v_{i+2}$ cannot be, as that would imply $v_i <_p^+ v_{i+2}$ and will contradict the induction hypothesis.

Similarly, if $(v_i, v_{i+1}), (v_{i+1}, v_{i+2})$, by the definition of E_c, $(v_i, v_{i+2}) \in E_c$ which again contradicts the induction hypothesis.

For convenience, when the ordering in $<_s$ is due to def-s(1), we will use $<_p^+$; otherwise, we will use $<_s$.

First, let us make the following observations for $v_i <_p^+ v_j$:

1. Either $r_j = \lambda_c(v_j) = \texttt{w(}p_j\texttt{,}a_j\texttt{,}d_j\texttt{)}$, for some p_j, a_j, d_j or the chain $v_i <_p v_{i_1} <_p v_{i_2} <_p \ldots v_{i_k} <_p v_j$ has the following property: $v_{i_k} <_p v_j$ is due to def-p(2); that is, $\lambda_c(v_{i_k}) = \texttt{w(}p_{i_k}\texttt{,}a_j\texttt{,}d_j\texttt{)}$ and $\lambda_c(v_j) = \texttt{r(}p_j\texttt{,}a_j\texttt{,}d_j\texttt{)}$; that is, $v_{i_k} = \omega(v_j)$.

2. Either $\lambda_c(v_i) = \texttt{w(}p_i\texttt{,}a_i\texttt{,}d_i\texttt{)}$ or the chain $v_i <_p v_{i_1} <_p v_{i2} <_p \ldots <_p v_j$ has the following property: $v_i <_p v_{i_1}$ is due to def-p(2); that is, $\lambda_c(v_i) = \texttt{r(}p_i\texttt{,}a_i\texttt{,}d_i\texttt{)}$, and $\lambda_c(v_{i_1}) = \texttt{w(}p_{i_1}\texttt{,}a_i\texttt{,}d_{i_1}\texttt{)}$, where $d_i \neq d_{i_1}$.

These are actually specific instances of the more general observations given prior to Lemma 4.1. We will refer to the first observation as obs2(1), to the second as obs2(2).

Let us prove that k cannot be 1, as a special case.

Without loss of generality, assume that $v_1 <_p^+ v_2 <_s v_1$ be such a chain. Let us further assume that we have $v_1 <_p v_{i_1} <_p v_{i_2} <_p \ldots <_p v_{i_m} <_p v_2$, for some m. By obs2(1), there are two cases to consider:

1. $r_2 = \lambda_c(v_2)$ is a write, that is, $r_2 = \texttt{w(}p_2\texttt{,}a_2\texttt{,}d_2\texttt{)}$. Since $v_2 <_s v_1$, we have $(v_2, v_1) \in E_c$. There are two possibilities:

   (a) $r_1 = \lambda_c(v_1) = \texttt{r(}p_1\texttt{,}a_1\texttt{,}d_1\texttt{)}$, $a_1 \neq a_2$. Note that, $p_1 = p_2$ as $(v_2, v_1) \in E_c$. By obs2(2), we have $s_1 = \lambda_c(v_{i_1}) = \texttt{w(}p_{i_1}\texttt{,}a_1\texttt{,}d_{i_1}\texttt{)}$. We also have $\omega(v_1) <_p v_1$ by def-p(2). Assuming $\omega(v_1)$ is $\texttt{w(}p'\texttt{,}a_1\texttt{,}d_1\texttt{)}$ for some p' and by obs(1), we get

   $$\texttt{w(}a_1\texttt{,}d_1\texttt{)} <_{t'} \texttt{w(}a_1\texttt{,}d_{i_1}\texttt{)} <_{t'} \texttt{w(}a_2\texttt{,}d_2\texttt{)}$$

   Now note that for $(v_2, v_1) \in E_c$, we have the following constraint in $\mathcal{CS}_c$

   $$\texttt{w(}a_1\texttt{,}d_1\texttt{)} \prec \texttt{w(}a_1\texttt{,}d_{i_1}\texttt{)} \prec \texttt{w(}a_2\texttt{,}d_2\texttt{)}$$

   This constraint, however, is not satisfied by $<_{t'}$, which contradicts the assumption about $<_{t'}$ satisfying $\mathcal{CS}_c$. So this case is not possible.

(b) $r_1 = \lambda_c(v_1) = \mathtt{r}(p_1,a_2,d_1)$. Let $v_3 = \omega(v_1) = \mathtt{w}(p',a_2,d_1)$, for some p'. Then, the following holds:

$$\mathtt{w}(a_2,d_1) \prec \mathtt{w}(a_2,d_2) \in \mathcal{CS}_c$$

Then, since $<_{t'}$ satisfies $\mathcal{CS}_c$ and by def-p(1), we have $v_2 <_p v_3$. But, we also have $v_3 <_p v_1 <_p^+ v_2$, which implies $v_3 <_p^+ v_2$. This contradicts the irreflexivity of $<_p^+$. So this case is not possible.

(c) $r_1 = \lambda_c(v_1) = \mathtt{w}(p_1,a_1,d_1)$. This implies that

$$\mathtt{w}(a_1,d_1) \prec \mathtt{w}(a_2,d_2) \in \mathcal{CS}_c$$

Since $<_{t'}$ satisfies $\mathcal{CS}_c$ and by def-p(1), we have $v_2 <_p v_1$. We also had $v_1 <_p^+ v_2$ by assumption. This contradicts the irreflexivity of $<_p^+$. So, this case is not possible.

2. $r_2 = \lambda_c(v_2)$ is a read, that is, $r_2 = \mathtt{r}(p_2,a_2,d_2)$. There are three possibilities:

(a) $r_1 = \lambda_c(v_1)$ is a read to a different address; that is, $r_1 = \mathtt{r}(p_1,a_1,d_1)$. Then, by obs(2), we cannot have $v_1 <_p v_2$. Then, by obs2(1), we have $v_1 <_p^+ v_{i_m} <_p v_2$ and $\lambda_c(v_{i_m}) = \mathtt{w}(p',a_2,d_2)$, for some p'. Also, by obs2(2), we have $v_1 <_p v_{i_1} <_p^+ v_2$, where $\lambda_c(v_{i_1}) = \mathtt{w}(p_{i_1},a_1,d_{i_1})$, such that $v_{i_1} \neq v_{i_m}$. Let $v_3 = \omega(v_1)$. By def-p(2), we have $v_3 <_p v_1$. Combining all, we get $v_3 <_p v_{i_1} <_p v_2$, which implies

$$\mathtt{w}(a_1,d_1) <_{t'} \mathtt{w}(a_1,d_{i_1}) <_{t'} \mathtt{w}(a_2,d_2)$$

By the definition of $\mathcal{CS}_c$, we have

$$\mathtt{w}(a_1,d_1) \prec \mathtt{w}(a_1,d_{i_1}) \prec \mathtt{w}(a_2,d_2) \in \mathcal{CS}_c$$

This contradicts the fact that $<_{t'}$ satisfies $\mathcal{CS}_c$. So, this case is not possible.

(b) $r_1 = \lambda_c(v_1)$ is a read to the same address; that is, $r_1 = \texttt{r(}p_1\texttt{,}a_2\texttt{,}d_1\texttt{)}$. Then, the following holds:

$$\texttt{w(}a_2\texttt{,}d_1\texttt{)} \prec \texttt{w(}a_2\texttt{,}d_2\texttt{)} \in \mathcal{CS}_c$$

Let us assume that $v_3 = \omega(v_1)$ and $v_4 = \omega(v_2)$. Then, since $<_{t'}$ satisfies $\mathcal{CS}_c$ and by def-p(1), we have $v_4 <_p v_3$. That, in turn, implies $v_4 <_p v_2 <_p v_3 <_p v_1$. But, we get $v_2 <_p^+ v_1$ and, by assumption, we have $v_1 <_p^+ v_2$. That contradicts the irreflexivity of $<_p^+$. So, this case is not possible.

(c) $r_1 = \lambda_c(v_1)$ is a write; that is, $r_1 = \texttt{w(}p_1\texttt{,}a_1\texttt{,}d_1\texttt{)}$. Then, the following holds:

$$\texttt{w(}a_1\texttt{,}d_1\texttt{)} \prec \texttt{w(}a_2\texttt{,}d_2\texttt{)} \in \mathcal{CS}_c$$

Let $v_3 = \omega(v_2)$. Then, since $<_{t'}$ satisfies $\mathcal{CS}_c$ and by def-p(1), we have $v_3 <_p v_1$. By obs2(1), we have $v_1 <_p^+ \ldots <_p v_3 <_p v_2$. This implies $v_1 <_p^+ v_3$, which is a contradiction. So this case is not possible.

Therefore, the chain cannot be of length 2. For the remainder of the proof, we are assuming that $k \geq 2$.

Let us turn back to the general case, $v_1 <_s v_2 <_s \ldots v_{k+1} <_s v_1$. Without loss of generality, assume that $v_{k+1} <_p^+ v_1$. Hence, $(v_1, v_2) \in E_c$. For the following, assume that r_j denotes $\lambda_c(v_j)$ for any $j \in [k+1]$. There are five cases to consider:

1. r_1 and r_2 are reads to different addresses; that is, $r_1 = \texttt{r(}p\texttt{,}a_1\texttt{,}d_1\texttt{)}$ and $r_2 = \texttt{r(}p\texttt{,}a_2\texttt{,}d_2\texttt{)}$, and $a_1 \neq a_2$. Let $w_1 = \omega(v_1)$. By obs2(1), one of the following holds:

   (a) $v_{k+1} <_p^+ w_1 <_p v_1$, or

   (b) $v_{k+1} = w_1 <_p v_1$.

   By obs2(2), there is a write w_3 to a_2 with $\lambda_c(w_3) = \texttt{w(}p'\texttt{,}a_2\texttt{,}d_3\texttt{)}$ for some $p' \in P$, such that either $v_2 <_p w_3 <_p^+ v_3$, or $v_2 <_p w_3 = v_3$.

Let $w_2 = \omega(v_2)$. We also have $w_2 <_p v_2$. Since $(v_1, v_2) \in E_c$, we have

$$\mathtt{w}(a_2\mathtt{,}d_2\mathtt{)} \prec \mathtt{w}(a_2\mathtt{,}d_3\mathtt{)} \prec \mathtt{w}(a_1\mathtt{,}d_1\mathtt{)} \in \mathcal{CS}_c$$

Since $<_{t'}$ satisfies $\mathcal{CS}_c$ and by def-p(1), one of the following must hold:

(a) $w_3 <_p w_2$. Note that, since $v_2 <_p^+ w_3$, we must have $w_2 <_p^+ w_3$. Both cannot hold, so this case is not possible.

(b) $w_1 <_p w_3$. By the above arguments, v_{k+1} is either w_1 or $v_{k+1} <_p^+ w_1$ holds. Similarly, w_3 is either v_3 or $w_3 <_p^+ v_3$ holds. Combining, we get $v_{k+1} <_p^+ v_3$, which results in $v_3 <_s v_4 <_s \dots v_{k+1} <_s v_3$, contradicting the induction hypothesis. So, this case is not possible.

Therefore, r_1 and r_2 cannot be both reads.

2. $r_1 = \mathtt{w(}p\mathtt{,}a_1\mathtt{,}d_1\mathtt{)}$ and $r_2 = \mathtt{r(}p\mathtt{,}a_2\mathtt{,}d_2\mathtt{)}$, for $a_1 \neq a_2$. This is proved similar to the previous case.

3. $r_1 = \mathtt{r(}p\mathtt{,}a_1\mathtt{,}d_1\mathtt{)}$ and $r_2 = \mathtt{w(}p\mathtt{,}a_2\mathtt{,}d_2\mathtt{)}$, for $a_1 \neq a_2$. Since $(v_1, v_2) \in E_c$, we have the following:

$$\mathtt{w}(a_2\mathtt{,}d_2\mathtt{)} \prec \mathtt{w}(a_1\mathtt{,}d_1\mathtt{)} \in \mathcal{CS}_c$$

Let $w_1 = \omega(v_1)$. Since $<_{t'}$ satisfies $\mathcal{CS}_c$ and by def-p(1), we must have $w_1 <_p v_2$. That implies that v_{k+1} is either w_1 or $v_{k+1} <_p^+ w_1$. Then, we have $v_{k+1} <_p^+ v_2$, which results in $v_2 <_s v_3 <_s \dots <_s v_{k+1} <_s v_2$, contradicting the induction hypothesis. So this case is not possible.

4. $r_1 = \mathtt{w(}p\mathtt{,}a_1\mathtt{,}d_1\mathtt{)}$ and $r_2 = \mathtt{w(}p\mathtt{,}a_2\mathtt{,}d_2\mathtt{)}$. Then, the following holds:

$$\mathtt{w}(a_2\mathtt{,}d_2\mathtt{)} \prec \mathtt{w}(a_1\mathtt{,}d_1\mathtt{)} \in \mathcal{CS}_c$$

Since $<_{t'}$ satisfies $\mathcal{CS}_c$ and by def-p(1), we have $v_1 <_p v_2$. But that would mean $v_{k+1} <_p^+ v_2$, resulting in $v_2 <_s v_3 <_s \dots v_{k+1} <_s v_2$, contradicting the induction hypothesis. So this case is not possible.

5. $r_1, r_2 \in \Sigma_a^A$ for some a. There are two possibilities:

   (a) $r_1, r_2 \in \Sigma_d^D$, for some d. Then, by previous arguments, $v_{k+1} <_p^+ v_1$ implies $v_{k+1} <_p^+ v_2$, resulting in $v_2 <_s v_3 <_s \ldots v_{k+1} <_s v_2$, contradicting the induction hypothesis.

   (b) r_1 and r_2 have different data values. By def-p(2) and the transitivity of $<_p^+$, r_1 and r_2 are ordered by $<_p^+$. Either ordering would result in a cycle of length less than $k+1$, contradicting the induction hypothesis. So, this case is not possible.

This covers all possible cases all of which result in contradiction. We, therefore, conclude that the claim of the lemma holds.

□

Hence, $<_s^+$ is indeed a strict partial order.

The final step in the proof is to construct a total order $<_{sc}$ consistent with $<_s^+$. That such a total order exists follows from the fact that $<_s^+$ is a strict partial order.

Definition 4.4 *Let $<_{sc}$ be a total order over V_c such that $v_1 <_s^+ v_2$ implies $v_1 <_{sc} v_2$.*

As the name hints, $<_{sc}$ is actually a total order that satisfies all the requirements of interleaved-sequentiality.

Lemma 4.3 *$<_{sc}$ satisfies all the requirements of interleaved-sequentiality.*

Proof (Lemma 4.3): Assume that it does not. Let us consider the possible violations.

1. (Case 1(a)): Assume that, we have $v_1 <_{sc} v_0$, for $v_0, v_1 \in V_c$ such that $\lambda_c(v_0) \in \Sigma_0^D$, $\lambda_c(v_1) \in \Sigma_d^D$, for some $d \neq 0$, and $\lambda_c(v_0), \lambda_c(v_1) \in \Sigma_a^A$. Let $w_1 = \omega(v_1)$. By definition of $\mathcal{CS}_c$ and $<_t$, and def-p(1), we must have $v_0 <_{sc} w_1$. We also have either $w_1 = v_1$ or $w_1 <_{sc} v_1$. Combining, we get $v_0 <_{sc} v_1$, contradicting the fact that $<_{sc}$ is a total order. So, this case is not possible.

2. (Case 1(b)): Assume that, we have $v_1 <_{sc} v_2 <_{sc} v$ such that $\lambda_c(v_1) \in W \cap \Sigma_a^A \cap \Sigma_d^D$ for some a, d, $\lambda_c(v_2) \in \Sigma_a^A \cap \Sigma_{d_2}^D$ for some $d_2 \neq d$, and $\lambda_c(v) \in R \cap \Sigma_a^A \cap \Sigma_d^D$. Note that, v_1 and v_2 are necessarily ordered by $<_p$. If $v_1 <_{sc} v_2$, then we must have $v_1 <_p v_2$; otherwise, we would not have an irreflexive $<_p^+$. By def-p(2), we also have $v <_p v_2$. But that is a contradiction. So, this case is not possible.

3. (Case 2): Assume that $(v_1, v_2) \in E_c$ and $v_2 <_{sc} v_1$. By the definition of $<_s$ we have $v_1 <_s v_2$. This is a contradiction. So, this case is not possible.

Therefore, none of the requirements of interleaved-sequentiality can be violated by $<_{sc}$.

□

Combining the results of all the above lemmas, we conclude that the concurrent execution is interleaved-sequential.

□

4.5 Minimal Sets

In the previous section, we proved that interleaved-sequentiality checking can be reduced to an equivalent problem of constraint satisfaction. In this section, we will make use of this new formulation.

Previous work on interleaved-sequentiality checking either completely ignored the problem of finding the subset of the execution that violated the property [22], or tried to characterize it in terms of cycles [50]. With the constraint sets, we can define what it means to have a minimal subset of a noninterleaved-sequential (non-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. 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 [50] or the one we explained in Section 4.3 due to [53].

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`

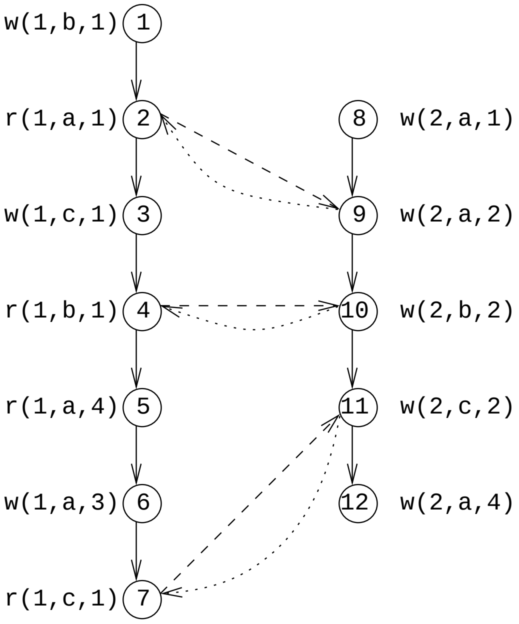


Figure 4.7. Sample non-i-s concurrent execution G_3 illustrating cycles and the minimal set. The dashed lines are the result of ordering w(a,1) before w(a,2). The dotted lines are the result of ordering w(a,4) before w(a,3).

(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 are the implied edges by the ordering of 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 4.5 *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 that itself is unsatisfiable but any proper subset of it is not.*

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. What we actually need is a collection of vertices whose constraints form a minimal set. Let us modify the κ and ι functions of the previous section. Define $\kappa'(v_1, v_2) = \{((C, (v_1, v_2)) \mid C \in \kappa(v_1, v_2)\}$, and $\iota'(v) = \{(C, v) \mid C \in \iota(v)\}$. That is, we are pairing each constraint with the vertex or vertices that are the causes of the constraint. For a concurrent execution G_c, let the *augmented constraint set* $\mathcal{CA}_c$ be the set $(\bigcup_{e \in E_c} \kappa'(e)) \cup (\bigcup_{v \in V_c} \iota'(v))$. We say that $\mathcal{CA}_c$ is *satisfiable* if and only if $\sigma(\mathcal{CA}_c) = \{C \mid \exists x \in E_c \cup V_c, (C, x) \in \mathcal{CA}_s\}$ is satisfiable.

For any $C'_1 = (C_1, (v_1, v_2)) \in \mathcal{CA}_s$, $C'_2 = (C_2, v) \in \mathcal{CA}_s$ and $v, v_1, v_2 \in V_c$, define $\nu_e(C'_1) = \{v_1, v_2, \omega(v_1), \omega(v_2)\}$, $\nu_v(C'_2) = \{v, \omega(v)\}$. Let $\nu(C') = \nu_e(C')$ if $C' \in dom(\nu_e)$; $\nu(C') = \nu_v(C')$ if $C' \in dom(\nu_v)$. We extend the definition of minimal constraint set to augmented constraint sets in the obvious way and call it the *minimal augmented constraint set.*

Definition 4.6 *Let G_c be a non-i-s concurrent execution and C_{min} be a minimal augmented constraint set of $\mathcal{CA}_c$. Then, the set $V_{min} = \{\nu(C) \mid C \in C_{min}\}$ is called the minimal instruction set.*

Example 5 *It can be readily verified that G_3 of Fig. 4.7 has a single minimal instruction set which is equal to V_3.*

Example 6 *It turns out that V_f of G_f is a minimal instruction set as well. To demonstrate this, let us consider $G_{f'}$ where $V_{f'} = V_f \setminus \{16\}$ and $E_{f'}$ is E_f restricted to $V_{f'}$. Then,*

$$\begin{aligned} \mathcal{CS}_{f'} = \quad \mathcal{CS}_f \setminus \ & \{\texttt{w}(c,\texttt{1}) \prec \texttt{w}(c,\texttt{2}) \prec \texttt{w}(b,\texttt{2}), \\ & \ \texttt{w}(c,\texttt{1}) \prec \texttt{w}(c,\texttt{2}) \prec \texttt{w}(a,\texttt{1})\} \end{aligned}$$

Let us define $<_t$ to be the total order

$$\texttt{w(}c\texttt{,1)} <_t \texttt{w(}c\texttt{,2)} <_t \texttt{w(}b\texttt{,1)} <_t \texttt{w(}a\texttt{,2)} <_t \texttt{w(}b\texttt{,2)} <_t \texttt{w(}a\texttt{,1)}$$

Then, it can be readily verified that $<_t$ satisfies $\mathcal{CS}_{f'}$. We conclude that V_f is a minimal instruction set after trying all the other vertices as we did for `16` *above.*

For a constraint $x \prec y \prec z$, we say that x appears in the *first position*, y in the *middle position* and z in the *last position*. Similarly, in the constraint $x \prec y$, x appears in the first position, y in the last.

We need one more result to conclude this section. We have to show that the constraint set built out of a minimal instruction set contains the minimal augmented constraint set that was used for constructing the minimal instruction set.

Lemma 4.4 *Let C_m be a minimal constraint set and $x \prec y \prec z$ be in C_m. Then, there exists at least one constraint in C_m where y appears either in the first position or the last position.*

Proof (Lemma 4.4): Assume the contrary. Then all the constraints in which y appears are of the form $x_1 \prec y \prec z_1$. Note that, by the definition of κ, x_1 and y must be writes to the same address and z_1 must be a write to different address. By the definition of C_m, $C'_m = C_m \setminus \{x \prec y \prec z\}$ is satisfiable. Furthermore, the total order $<_t$ that satisfies C'_m must have $x <_t y <_t z$ since any other ordering will satisfy C_m contradiction the unsatisfiability of C_m.

Now, let us assume that y is $\texttt{w(}a\texttt{,}d\texttt{)}$, for some $a \in A$, $d \in D$. Let x_{min} be of the form $\texttt{w(}a\texttt{,}d_{min}\texttt{)}$ such that for any $x' = \texttt{w(}a\texttt{,}d'\texttt{)}$, $d' \neq d_{min}$, we have $x_{min} <_t x'$. Let $<_n$ be defined as follows:

1. $y <_n x_{min}$.

2. $x' <_t x_{min}$ implies $x' <_n y$.

3. $x' <_t z'$, $x', z' \neq y$ imply $x' <_n z'$.

By the previous argument, there must be at least one constraint in C'_m that $<_n$ does not satisfy. Such a constraint cannot be one in which y does not appear as $<_t$ satisfies it and $<_n$ agress with $<_t$ on $\mathbf{elem}(C'_m) \setminus \{y\}$. So, the constraint must be of the form $x_1 \prec y \prec z_1$. Now, note that if $x_1 \neq x_{min}$, then $x_{min} <_t x_1$. Then, by definition of $<_n$, $y <_n x_1$. But this satisfies the constraint, contradicting our assumption ($<_n$ satisfies C_m, but C_m was assumed to be unsatisfiable). Therefore, y must appear in either the first or the last position of a constraint in C_m.

□

Corollary 4.1 *If a term* $\mathtt{w(}a\mathtt{,}d\mathtt{)}$ *appears in a minimal augmented constraint set,* $\omega_c^{-1}(\mathtt{w(}a\mathtt{,}d\mathtt{)})$ *is in the corresponding minimal instruction set.*

Proof (Corollary 4.1): By the previous lemma, we know that any write has to appear either in the first or the last position of a constraint. By the definition of κ' and ι', all such writes along with the instructions that cause the constraint, will be included in the instruction set.

□

Theorem 4.2 *Let* V_{min} *be a minimal instruction set for a concurrent execution* G_c*. Then,* $(V_{min}, E_c \cap V_{min}^2)$ *is a non-i-s concurrent execution.*

Proof (Theorem 4.2): Follows from Corollary 4.1 and the definition of minimal augmented constraint set.

□

Example 7 *Let us examine the concurrent execution* G_d *given in Fig. 4.8.*

There are two minimal instruction sets in G_d*:*

1. $V_{min}^1 = \{\mathtt{1}, \mathtt{5}, \mathtt{6}, \mathtt{8}\}$

2. $V_{min}^2 = \{\mathtt{3}, \mathtt{4}\}$

If we consider the augmented constraint set $\mathcal{CA}_d$ *for* G_d*, we will have* $((\mathtt{w(}c\mathtt{,2)} \prec \mathtt{w(}c\mathtt{,2)}), (\mathtt{3}, \mathtt{4}))$ *in* $\mathcal{CA}_d$*. The constraint* $\mathtt{w(}c\mathtt{,2)} \prec \mathtt{w(}c\mathtt{,2)}$ *is never satisfiable. So,*

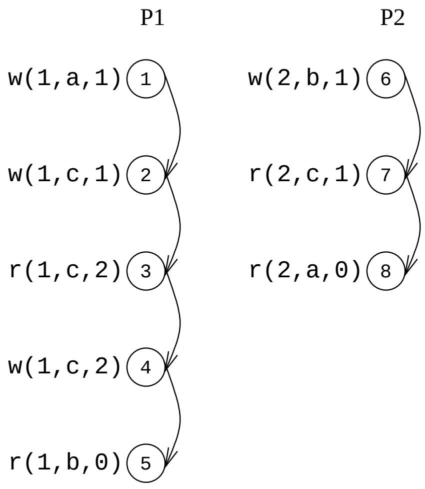


Figure 4.8. The concurrent execution G_d with two minimal instruction sets.

that constraint alone will give us a minimal instruction set, which happens to be equal to V^2_{min}.

4.6 Finiteness Result for Unambiguous Executions

In this section, we will show that we need to check only a bounded number of concurrent executions to conclude that a system cannot generate unambiguous non-i-s concurrent executions.

Lemma 4.5 *Let P, A, D be all finite. Then the size of any minimal instruction set of any non-i-s unambiguous concurrent execution is bounded.*

Proof (Lemma 4.5): Let G_c be a non-i-s concurrent execution and $\mathcal{CA}_c$ be its augmented constraint set. Let $k_w = |A| \times (|D|+1)$. Observe that there are at most k_w different (write) terms that can appear in $\sigma(\mathcal{CA}_c)$. The number of different constraints of three terms is then bounded by $k_w \times |D| \times (k_w - (|D|+1))$. Similarly, the number of two termed constraints is bounded by k_w^2. The size of any minimal augmented constraint set, C_m, is also bounded as $(C_1, x), (C_2, y) \in C_m$ implies $C_1 \neq C_2$. This in turn means that the minimal instruction set is bounded since there can be at most $4k_c$ vertices, where k_c is the cardinality of C_m which is less than $k_w^2(|D|+1)$.

□

It is worth noting that since the constraints do not take the processor index into account, the bound does not depend on the number P.

The problem of sequential consistency checking is to verify for the finite state machine modelling an smi whether all its runs are i-s. We call the (finite-state) machine to be verified the *implementation*.

At this point, let us relate the formalization used so far in this chapter to the formalization we introduced in Chapter 2. The labels of the vertices of concurrent executions are in a one-to-one correspondence with $\mathcal{O}^{RW}$. As alluded to before, a label of the form $\mathtt{r}(p,a,d)$ in a concurrent exectution corresponds to the symbol $(\mathtt{r}_\circ,p,a,d)$. Similarly, $\mathtt{w}(p,a,d)$ corresponds to the symbol $(\mathtt{w}_\circ,p,a,d)$. Let us denote this correspondence by *cor*. Now, for a given input/output stream pair $\sigma = ((\mathbf{p},\mathbf{n}),(\mathbf{q},\mathbf{m}))$,[11] there is precisely one concurrent execution $G_\sigma = (V_\sigma, E_\sigma, \lambda_\sigma)$ defined as follows:

1. $V_\sigma = [|\mathbf{q}|]$.

2. $\lambda_\sigma(v) = cor(q_j)$, when $\widetilde{\eta}(v) = j$. That is, the vertex v has the label corresponding to the response of the v^{th} input symbol.

3. $(v_1, v_2) \in E_\sigma$ if and only if p_{v_1} and p_{v_2} belong to the same processor and $v_1 < v_2$. That is, an edge from v_1 to v_2 of the concurrent execution exists if and only if the instruction that caused the response $\lambda_\sigma(v_1)$ precedes that of $\lambda_\sigma(v_2)$ in input/time and both belong to the same processor.

As an illustration, consider the temporal ordering of instructions, denoted by i_j, and responses, where the response of instruction i_j is denoted by r_j, and the

[11] We require that $|\mathbf{p}| = |\mathbf{q}| = |\mathbf{n}|$, $\mathbf{n}$ and $\mathbf{m}$ be compatible, the mapping from input to output symbols be given by the normal computation $\widetilde{\eta}$ and whenever $\eta i = j$ we have $\rho^{RW}(q_j) = p_i$. Without the first and last requirements, the concurrent execution is not well-defined. The other (second and third) requirements are mainly for convenience and not essential.

corresponding execution depicted in Fig. 4.9. The direction of the dotted arrowed edges gives the temporal ordering; i_1 is the first, i_4 the second, r_4 is the last. In the sequential execution of processor 1, P_1, we have $(r_1, r_2), (r_2, r_3) \in E_1$, even though temporally r_3 precedes r_2, because we have i_1 (temporally) precede i_2 which in turn (temporally) precedes i_3. Note also that any temporal ordering between instructions belonging to different processors is not represented in the concurrent execution.

Lemma 4.6 *Let G_c be a non-i-s unambiguous concurrent execution of an implementation, $\mathcal{CA}_c$ be its augmented constraint set, C_m be a minimal augmented constraint set and V_m be the corresponding minimal instruction set. Then, the same C_m can be generated by a run that does not visit any state of the implementation more than $2|V_m| + 1$ times.*

Proof (Lemma 4.6): Let $\mathbf{r}$ be the run of the implementation that generated G_c (see Fig. 4.10). Let the states $s_{i_j}, j \in [2|V_m|]$ be the states at which either a response $\lambda_c(v)$ for a $v \in V_m$ is generated or the instruction of that response is input,

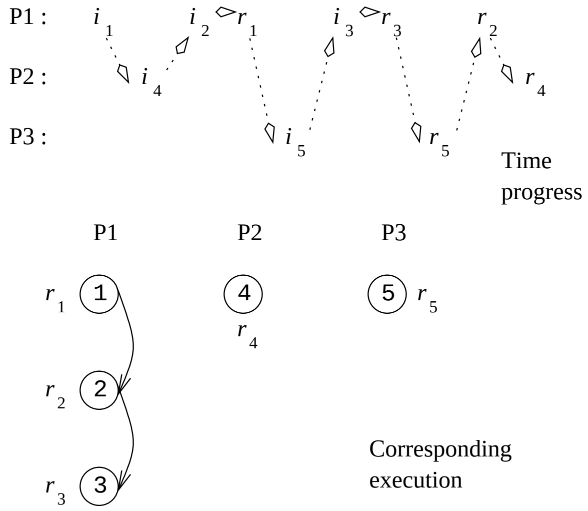


Figure 4.9. The relation between the temporal ordering of instructions/responses and its associated corresponding concurrent execution; the top half gives the temporal ordering.

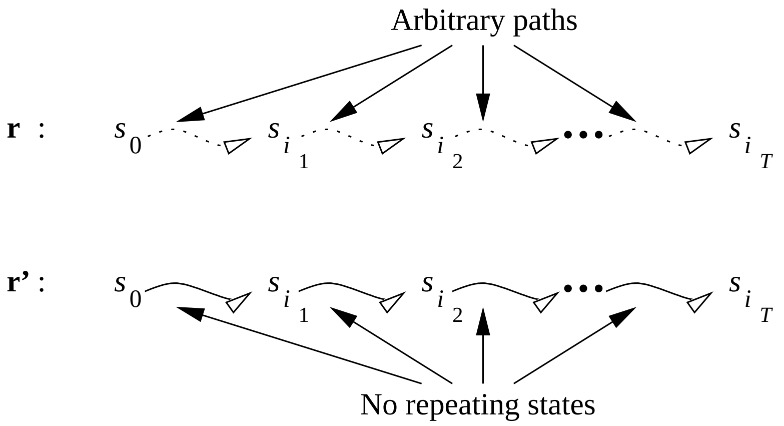


Figure 4.10. The arbitrary run $\mathbf{r}$ and the constructed run $\mathbf{r}'$, where $T = 2|V_m|+1$.

such that s_{i_k} temporally precedes s_{i_l} if and only if $k < l$. Let s_0 be the initial state of the run $\mathbf{r}$ and $s_{i_{2|V_m|+1}}$ denote the final state. Let $\mathbf{r}'$ be the run that starts from s_0 such that on the path between any s_{i_j-1} and s_{i_j} no state is visited twice. Since we are preserving the relative order of instructions whose responses form the set V_m, the set of constraints generated for the concurrent execution corresponding to $\mathbf{r}'$ will be a superset of C_m; hence, C_m will be a minimal set of this concurrent execution as well. By construction, $\mathbf{r}'$ does not visit any state more than $2|V_m|+1$ times.

□

Theorem 4.3 *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.*

Proof (Theorem 4.3): The only if direction is obvious. The if direction follows from the two previous lemmas.

□

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 previously. 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 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.

4.7 Summary

In this chapter, we have defined a new problem, constraint satisfaction, which we prove to be equivalent to the interleaved-sequentiality checking of a concurrent execution. We have formalized the notion of a crux of a non-i-s execution which is still non-i-s but any proper subset of it is i-s. Such a characterization of minimality was not possible before and it is a direct consequence of the constraint satisfaction problem. We further made use of this minimality definition to prove a strong result about the formal verification for sequential consistency: an implementation has a non-i-s unambiguous execution if and only if it has one of bounded size, bound being a function of the number of different addresses and data values of the implementation.

CHAPTER 5

CONCLUSION

In this final chapter, we will summarize the results presented in this dissertation and propose several topics as possible future work.

5.1 Summary

The beginning of this dissertation can be traced back to the reading of [12]. The undecidability result presented in that work had been used in succeeding works as evidence to the undecidability of the formal verification of sequential consistency even for finite state systems. Not quite convinced by that interpretation, further investigation led us to believe that this deduction was an artifact of the formalization used in the problem (see Appendix). Appealing to the intuition one had regarding a memory, we have then developed a new formalization. Similar formalizations, where, even though not explicitly stated, a memory is viewed as a transducer, had been previously suggested, but our approach has two novel aspects:

1. We differentiate between a shared memory model and a shared memory system. The former, to which we coin the name specification, is a set of program, execution pairs. It basically gives a set of possible executions for each syntactically correct program. Hence, it is a binary relation and its definition should not be behavioral. On the other hand, a shared memory system, which we call an implementation, will have a behavioral description. An implementation realizes a relation. The formal verification of a shared memory model for a shared memory system then becomes the inclusion of the relation realized by that system, the implementation, by the relation defined by that shared memory model, or specification.

2. As is common for any hardware system, the most basic mathematical structure to be used for an implementation is a finite-state automaton. However, since we are now dealing with relations over programs and executions, we semantically differentiate instructions, inputs to memory, from responses, outputs generated by the memory. We, therefore, slightly change the interpretation of a string generated by a memory system. A string, in our formalization, actually represents a combination of two substrings, one corresponding to a program, the other to its execution; hence, an element in a relation over programs and executions.

There is one problem that should not be overlooked: the mapping between instructions and responses per computation. That is, given a program, execution pair where the program might possibly have several identical instructions (for instance, same processor issuing several read queries for the same address) how can we tell which response, not necessarily identical, corresponds to which instruction?

In specifications, this problem is rather easily solved. Permutations, which can be represented as a string over natural numbers, can be used to represent this mapping.

In implementations, where a finite alphabet is required as we are dealing with finite state machines, using strings over the infinite set of natural numbers is not possible. The alternative to this, which seems to have been the popular approach taken in previous work [8, 12, 36, 38], is to assume certain orderings in implementation. For instance, if in order completion per processor is assumed, the mapping becomes trivial; a response is always mapped to the most recent pending instruction. Trying to generalize our results as much as possible, we have again appealed to the finiteness of the user that the memory interacts with. It is obvious that the instructions and responses should be tagged for any sort of mapping.[1] We argue that for finite users and memory systems, we can do away with arbitrary

[1] A mapping that assumes in-order completion would need only a single tag per processor; hence the index of the processor that issues the instruction will serve the purpose of a tag.

methods for tagging and without any loss of generality, deal with only a normal tagging method which we call normal coloring.

We have demonstrated our formalization by defining sequential consistency as a specification and lazy caching as an implementation. The ease with which we were able to define these makes us believe that this formalization can be used extensively in real world problems related to shared memories.

The next step was to formulate the problem of shared memory verification in our formalization. This has been extensively studied in Chapter 3. We tried to prove the sequential consistency of lazy caching using the definitions given in Chapter 2. Using a novel approach whereby we approximate a given shared memory model by an infinite hierarchy of (finite) implementations each of which is called a memory model machine, or in this context an SC machine, we present the problem as a regular language inclusion problem between two finite state automata whose strings are over a pair of alphabets. We obtained two noteworthy results in this endeavor:

1. There is a problem of fairness in the lazy caching protocol. This was glossed over in the original paper that introduced the lazy caching [8]. In other works where this problem was revisited, this point seems to have been ignored. The problem is due to the possibility of delaying an instruction for an unbounded amount of time. This nondeterminism causes the approximate approach to fail: for any given SC machine, there is always a computation which the lazy caching can perform but is not contained in the set of computations of that SC machine. So, in a sense, the SC machines implicitly impose a certain fairness. Assuming a finite yet undetermined bound, we were able to prove that the language of a lazy caching implementation is indeed contained in the relation of a certain SC machine. Considering that the undeterminedness of that value is some sort of abstraction for the temporal execution of the implementation, at some level that bound will have a determined value, corresponding to at least a greatest bound, and the problem will be mechanically solved by a tool that can perform regular language inclusion.

2. In the unfair case, where arbitrary delays are assumed, we can use a hypothetical and infinite SC machine with unbounded queues to prove that the lazy caching algorithm is sequentially consistent. This time, we will not be able to have a regular language inclusion but the proof still depends on an argument of language containment.

Another important aspect about the approach we propose for shared memory verification is that the method of approximating memory machines is amenable to improvement. Let us assume that we are given a memory model, S, and a memory implementation, I. Assume further that, for this implementation we cannot find any S machine whose language contains the language of I. If, through some other means, we are able to prove that I indeed satisfies S, then we can use I to refine the hierarchy of S machines: for any S machine, define the S' machine which is the union of I and that S machine. This refined hierarchy will be a better approximation for that memory model.

Changing the context a little, in Chapter 4, we have considered the problem of checking the interleaved-sequentiality of a single computation of a memory system. Any computation a sequentially consistent memory system performs is interleaved-sequential. Previous works on this topic were graph based; the interleaved-sequentiality of a computation was a property of a graph, or a set of graphs, that was defined by the execution of the computation. We have concentrated on the set of unambiguous executions in which a data value to an address is written at most one time. We have transformed this problem to an equivalent problem of constraint satisfaction. The constraint satisfaction problem is satisfied if and only if the execution on which it is based is interleaved-sequential. The structure of the constraint satisfaction problem gives more insight why a formulation based on binary relations, such as a graph, is not suitable. We then used this equivalent problem to define the essential part of a non-interleaved-sequential execution. This essential part is defined to be the minimal set of responses which still form a non-interleaved-sequential execution but any proper subset of it is interleaved-sequential. This pruning of the irrelevant responses from the execution was previously not as obvious. Finally, we were able to show that for the set of unambiguous executions in a finite implementation, there exists an execution that violates interleaved-sequentiality if and only if there exists one such execution of a computable bound, a bound that is a function of the data and address spaces of the implementation. This result is deceitfully simple, yet was not as easily obtainable with previous approaches.[2]

5.2 Future Work

One possible direction for further research on this subject is more or less obvious: the development of a tool. We have been talking about, or even advocating, how the problem of shared memory verification can be cast as a language inclusion problem. We have even defined an approximate method where the language inclusion is checked for two finite-state automata. It is obvious that these steps, once the implementation and the specification are given, can be efficiently solved following an algorithm. The tool we anticipate developing, hence, should have the following two major operations:

1. It should be able to transform a length-preserving and rational transducer from I to O to a finite-state automaton over $I \times O$. This operation is sometimes called the *synchronization* of the transducer.

2. It should be able to check regular language inclusion for any given pair of finite-state automata.

These operations, however, are not computationally cheap. For instance, the second operation is known to be PSPACE-complete for arbitrary finite-state automata. As is true for these kinds of problems, the specific nature of the problem might enable certain optimizations, optimizations that would not work for the general case. We hope that such optimizations will result in a tool that can be used efficiently by interested parties, such as the verification engineers in the industry or shared memory designers.

[2] We are not sure, at this point, whether this result can be obtained at all with previous approaches; the last part of this sentence should be read as cautionary rather than factual.

Another possible path for future research is rather theoretical. As we have seen, the memory model machine approach depends on a sufficient condition: if the language containment holds, then the implementation satisfies the memory model. But, the result is inconclusive in the case the containment does not hold.

Let S be a given memory model. Let $\mathcal{I}_n$ be the set of all implementations which have exactly n states and which satisfy S. Clearly, there are finitely many implementations of n states and, hence, $\mathcal{I}_n$ is a finite set of transducers. Let $\mathcal{I}'_n$ be the set, isomorphic to $\mathcal{I}_n$, such that each transducer of $\mathcal{I}_n$ is converted to a language equivalent finite-state automata (over pair of alphabets). Let I_n be the automaton whose language is equal to the union of the languages of all automata of $\mathcal{I}'_n$. Then, it is easy to see that an implementation of n states satisfies S if and only if its language is contained in that of I_n.

The argument above proves that for a given memory model S and a number n, there is always a finite-state machine which will generate all program, execution pairs each of which is generated by an n-state implementation satisfying S. Of course, that argument is circular; we have to first form the set $\mathcal{I}_n$ which will need a procedure to decide whether an n-state implementation satisfies S or not. However, the point we are trying to make is that such a machine *exists*.

We conjecture that a machine whose language includes the language of I_n can be effectively computed. We base this hypothesis on the intuition that a finite-state machine can only retain a finite amount of *information* and it should be always possible to reduce this information to a finite set of equivalent classes of *information templates*.

Much like the sufficiency result of the third chapter, the previous chapter has a necessity result. If the implementation does not have any unambiguous execution violating the memory model, we cannot conclude whether the implementation does indeed satisfy the memory model. Therefore, we think that it is more suitable to perceive that part as a possible debugging method for sequential consistency.

The number of repetitions of a state to be checked we have provided for that part is not tight. We have neglected certain simplifications in the constraint set, as our only goal was to show the decidability of the problem itself. Furthermore, certain assumptions, such as the symmetry of address or data spaces or even processors, can considerably reduce the bound we have given. We intend to work on this problem by making use of these observations. We hope that a much smaller bound on the number of repetitions of a state will be enough to make this approach a valuable and useful method for the debugging of sequentially consistent systems.

Coherence, which has been also defined as sequential consistency per address, can be debugged using the approach of Chapter 4. In fact, we believe that a graph approach is sufficient for coherence as constraints of the form $x \prec y \prec z$ will not appear in a constraint set constructed for coherence. Our initial work shows that the number of repetitions per state cannot be larger than $4p$, where p is the number of processors in the implementation. We hope to formalize these results in our (near) future work.

APPENDIX

EXECUTION BASED FORMALISM AND THE UNDECIDABILITY RESULT

In [12], a shared memory implementation is defined to be a finite state machine. Its alphabet consists of read and write *events*. The event $\texttt{r}(p,a,d)$ denotes the reading of data value d from address a by processor p. The event $\texttt{w}(p,a,d)$ denotes the writing of data value d into address a by processor p. We can think of these events as the responses generated by the memory; the instructions, which are the inputs to the memory and issued by the processors, are absent in this formalization. The language of the finite state machine characterizes the shared memory implementation.

It is claimed that in this framework, sequential consistency[43] can be defined as a property of the language of the finite state machine. Any string in the language of the finite state machine is treated as a trace, that is, a set of equivalent strings where equivalence is defined with respect to a certain dependence relation. Suffice it to say, at this point, that the dependence relation implicitly imposes a temporal relation between the order of issuing per processor and their completion (sometimes called commitment) times. If an instruction of a processor is issued before another instruction of the same processor, the completion (commitment) time of the former is assumed to be before that of the latter. This is reflected by the fact that the response corresponding to the former instruction will appear before the response corresponding to the latter instruction in the string representing the execution.

A string is *serial* if any read event of an address in the string returns the value of the rightmost write to the same address of the prefix up to this read, or the initial value in the absence of such a write. A string is sequentially consistent[1] if there exists at least one serial string in its equivalence class. A set of strings is sequentially consistent if all its members are. A finite-state machine is sequentially consistent if its language is.

This is a typical execution only definition. We are given a certain collection of executions, in this particular instance, a set of strings, and the shared memory model is defined as a property of this collection. However, we believe that this approach is inadequate. Consider the following program and its execution, given in the form of [12].

$$\texttt{w(1,}a\texttt{,2)}\ \texttt{r(1,}a\texttt{)}^*\ \texttt{r(2,}a\texttt{)}^*\ \texttt{w(2,}a\texttt{,1)}$$

$$\texttt{w(1,}a\texttt{,2)}\ \texttt{r(1,}a\texttt{,1)}^*\ \texttt{r(2,}a\texttt{,2)}^*\ \texttt{w(2,}a\texttt{,1)}$$

According to the definition of [12], the regular expression above and the finite-state machine generating it are sequentially consistent as any string in its language is serial. Let us assume that N is the cardinality of the state space of the finite-state machine. Think of the program where we issue $2N$ $\texttt{r(1,}a\texttt{)}$ instructions and $2N$ $\texttt{r(2,}a\texttt{)}$ instructions. By the execution string, we know that the first instruction to commit is the $\texttt{w(1,}a\texttt{,2)}$ instruction. This is followed by the commitment of one of the read instructions $\texttt{r(1,}a\texttt{)}$ with the value 1. 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 commits the read instruction before the issuing of the $\texttt{w(2,}a\texttt{,1)}$ instruction. If at the instant the machine commits this read instruction we stop feeding the finite-state machine with instructions, it will either terminate with a nonserial execution or it will hang waiting for the write instruction it *guessed*. Either case belongs to an implementation that is not sequentially consistent.

[1] In our terminology, interleaved-sequential.

2. The machine commits the read instruction after the issuing of the `w(2,`a`,1)` instruction. This means that the machine has not committed any instruction 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 responses. 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 that 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. This clearly does not correspond to a reasonable memory, let alone sequential consistency.

The basic fallacy here is the abstraction of input, or the program. An execution alone is not sufficient to characterize a memory implementation; it is only suitable for execution specific problems, as the problem presented in Chapter 4.

REFERENCES

[1] Adve, S. V. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, December 1993.

[2] Adve, S. V., and Gharachorloo, K. Shared memory consistency models: A tutorial. *IEEE Computer 29*, 12 (December 1996), 66–76.

[3] Adve, S. V., and Hill, M. D. Weak ordering - a new definition and some implications. Tech. Rep. 902, Computer Sciences, University of Wisconsin - Madison, December 1989.

[4] Adve, S. V., and Hill, M. D. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing* (August 1990), pp. 47–50.

[5] Adve, S. V., and Hill, M. D. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)* (May 1990), pp. 2–14.

[6] Adve, S. V., and Hill, M. D. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems 4*, 6 (June 1993), 613–624.

[7] Adve, S. V., Pai, V. S., and Ranganathan, P. Recent advances in memory consistency models for hardware shared memory systems. *Proceedings of the IEEE 87*, 3 (March 1999), 445–455.

[8] Afek, Y., Brown, G., and Merritt, M. Lazy caching. *ACM Transactions on Programming Languages and Systems 15*, 1 (January 1993), 182–205.

[9] Ahamad, M., Bazzi, R. A., John, R., Kohli, P., and Neiger, G. The power of processor consistency (extended abstract). In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures* (1993), ACM Press, pp. 251–260.

[10] Ahamad, M., Neiger, G., Burns, J. E., Kohli, P., and Hutto, P. W. Causal memory: Definitions, implementations and programming. Tech. Rep. GIT-CC-93/55, College of Computing, Georgia Institute of Technology, July 1994.

[11] Alur, R., and Henzinger, T. A. Finitary fairness. *ACM Transactions on Programming Languages and Systems 20*, 6 (1998), 1171–1194.

[12] Alur, R., McMillan, K., and Peled, D. Model-checking of correctness conditions for concurrent objects. In *Symposium on Logic in Computer Science* (1996), IEEE, pp. 219–228.

[13] Attiya, H., and Friedman, R. A correctness condition for high-performance multiprocessors. *SIAM Journal on Computing 27*, 6 (December 1998), 1637–1670.

[14] Attiya, H., and Welch, J. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems 12*, 2 (May 1994), 91–122.

[15] Berstel, J. *Transductions and Context-free Languages.* Teubner, 1979.

[16] Bingham, J. D., Condon, A., and Hu, A. J. Toward a decidable notion of sequential consistency. In *Proceedings of the 15th annual ACM Symposium on Parallel Algorithms and Architectures* (2003), ACM Press, pp. 304–313.

[17] Braun, T., Condon, A., Hu, A. J., Juse, K. S., Laza, M., Leslie, M., and Sharma, R. Proving sequential consistency by model checking. Tech. Rep. TR-2001-03, Dept. of Computer Science, Univ. of British Columbia, 2001.

[18] Brinksma, E. Cache consistency by design. *Distributed Computing 12*, 2-3 (1999), 61–74.

[19] Chatterjee, P. Formal specification and verification of memory consistency models of shared memory multiprocessors. Master's thesis, School of Computing, University of Utah, March 2002.

[20] Chatterjee, P., and Gopalakrishnan, G. Towards a formal model of shared memory consistency for intel itanium. In *Proceedings of the International Conference on Computer Design, ICCD'01* (September 2001), pp. 515–518.

[21] Chatterjee, P., and Gopalakrishnan, G. A specification and verification framework for developing weak shared memory consistency protocols. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design* (2002), Springer-Verlag, pp. 292–309.

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

[23] Condon, A. E., and Hu, A. J. Automatable verification of sequential consistency. In *13th Symposium on Parallel Algorithms and Architectures* (2001), ACM, pp. 113–121.

[24] de Melo, A. C. M. A. Defining uniform and hybrid memory consistency models on a unified framework. In *Proceedings of the 32nd Hawaii International Conference on System Sciences, Vol.VIII-Software Technology* (January 1999), pp. 270–279.

[25] Dill, D., Park, S., and Nowatzyk, A. G. Formal specification of abstract memory models. In *Research on Integrated Systems : Proceedings of the 1993 Symposium* (March 1993), MIT Press, pp. 38–52.

[26] Dubois, M., and Scheurich, C. Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer 21*, 2 (February 1988), 9–21.

[27] Dubois, M., Scheurich, C., and Briggs, F. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture* (June 1986), pp. 434–442.

[28] Gao, G. R., and Sarkar, V. Location consistency: stepping beyond the barriers of memory coherence and serializability. Tech. Rep. ACAPS-78, ACAPS Laboratory, School of Computer Science, McGill University, December 1994.

[29] Gharachorloo, K., Adve, S. V., Gupta, A., Hennessy, J. L., and Hill, M. D. Specifying system requirements for memory consistency models. Tech. Rep. CSL-TR-93-594, Computer System Laboratory, Stanford University, 1993.

[30] Gibbons, P. B., and Merritt, M. Specifying nonblocking shared memories (extended abstract). In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures* (1992), ACM Press, pp. 306–315.

[31] Gibbons, P. B., Merritt, M., and Gharachorloo, K. Proving sequential consistency of high-performance shared memories (extended abstract). In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures* (1991), ACM Press, pp. 292–303.

[32] Goodman, J. R. Coherency for multiprocessor virtual address caches. In *Proceedings of the 2nd ASPLOS* (1987), pp. 72–81.

[33] Gopalakrishnan, G. A formalization of test-model checking, completeness results and case studies. In *Workshop on Advances in Verification* (2000).

[34] Graf, S. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Distributed Computing 12*, 2-3 (1999), 75–90.

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

[36] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 463–492.

[37] Higham, L., Kawash, J., and Verwaal, N. Defining and comparing memory consistency models. In *Proceedings of the 10th International Symposium on Parallel and Distributed Computing Systems* (October 1997), pp. 349–356.

[38] Hojati, R., Mueller-Thuns, R., Loewenstein, P., and Brayton, R. K. Automatic verification of memory systems which service their requests out of order. In *Proceedings of the ASP-DAC'95* (1995), pp. 623–630.

[39] Janssen, W., Poel, M., and Zwiers, J. The compositional approach to sequential consistency and lazy caching. *Distributed Computing 12*, 2-3 (1999), 105–127.

[40] Jonsson, B., Pnueli, A., and Rump, C. Proving refinement using transduction. *Distributed Computing 12*, 2-3 (1999), 129–149.

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

[42] Ladkin, P., Lamport, L., Olivier, B., and Roegel, D. Lazy caching in tla+. *Distributed Computing 12*, 2-3 (1999), 151–174.

[43] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers 28*, 9 (September 1979), 690–691.

[44] Lamport, L., Perl, S., and Weihl, W. When does a correct mutual exclusion algorithm guarantee mutual exclusion? *Information Processing Letters 76*, 3 (2000), 131–134.

[45] Landin, A., Hagersten, E., and Haridi, S. Race-free interconnection networks and multiprocessor consistency. In *Proceedings of the 18th International Symposium on Computer Architecture* (1991), ACM Press, pp. 106–115.

[46] Laza, M., Sharma, R., Condon, A., and Hu, A. J. Protocols for which proving sequential consistency is easy. Presented in the Workshop on Formal Specification and Verification Methods for Shared Memory Systems, October 2000.

[47] Lowe, G., and Davies, J. Using csp to verify sequential consistency. *Distributed Computing 12*, 2-3 (1999), 91–103.

[48] Mizuno, M., Raynal, M., and Zhou, J. Z. Sequential consistency in distributed systems: theory and implementation. Tech. Rep. 871, IRISA, October 1994.

[49] Mosberger, D. Memory consistency models. Tech. Rep. 93/11, Department of Computer Science, University of Arizona, 1993.

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

[51] Nalumasu, R., Ghughal, R., Mokkedem, A., and Gopalakrishnan, G. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In *Proceedings of the 10th International Conference on Computer-aided Verification (CAV)* (1998), pp. 464–476.

[52] Park, S., and Dill, D. L. An executable specification, analyzer and verifier for rmo (relaxed memory order). In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures* (July 1995), pp. 34–41.

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

[54] Raynal, M., and Schiper, A. A suite of formal definitions for consistency criteria in distributed shared memories. Tech. Rep. 968, Institut de Recherche en Informatique et Systèmes Aléatoires, IRISA, November 1995.

[55] Sakarovitch, J. *Éléments de Théorie des Automates.* Les Classiques de l'Informatique. Vuibert Informatique, September 2003.

[56] Scheurich, C., and Dubois, M. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture* (June 1987), pp. 234–243.

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

[58] Shi, W., Hu, W., and Tang, Z. An interaction of coherence protocols and memory consistency models in dsm systems. *Operating Systems Review 31*, 4 (1997), 41–54.

[59] Sorin, D. J., Plakal, M., Hill, M. D., and Condon, A. E. Lamport clocks: Reasoning about shared memory correctness. Tech. Rep. CS-TR-1998-1367, Computer Science Department, University of Wisconsin - Madison, 1998.

[60] Steinke, R. C., and Nutt, G. J. A unified theory of shared memory consistency. Obtained from ftp://ftp.cs.colorado.edu/pub/distribs/Nutt/jacm04.ps (to appear in the Journal of the ACM).

[61] Weiwu, H., and Peisu, X. Out-of-order execution in sequentially consistent shared-memory systems: theory and experiments. *ACM SIGARCH Computer Architecture News 25*, 4 (September 1997), 3–10.

[62] Zucker, R. N. Relaxed consistency and synchronization in parallel processors. Tech. Rep. 92-12-05, Department of Computer Science and Engineering, University of Washington, December 1992.