Model Checking Sequential Consistency and Parameterized Protocols

by

Jesse Bingham

B.Sc. University of Victoria, 1998
M.Sc. University of Victoria, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

August 2005

© Jesse Bingham, 2005

Abstract

Perhaps the most difficult aspect of designing a shared memory multiprocessor is the hardware protocol that facilitates the sharing of memory by multiple processors; these protocols are thus a natural target for formal verification. In this thesis we consider several problems relevant to model checking these protocols.

The ultimate specification of a protocol is the *memory model*. Our more theoretical contributions relate to the problem of model checking a protocol for the well-known memory model *sequential consistency* (SC). We define a restricted version of SC called *decisive SC* (DSC), which rules out pathologies admitted by SC, and explore the complexities of its verification problems. Our key results are that DSC of a single behavior is NP-complete, DSC of a protocol is PSPACE-hard, a bounded variant DSC_k is decidable in EXPSPACE, and full SC remains undecidable even when we require protocol behaviors to be prefix-closed. Also, we show that SC in conjunction with the ubiquitous property *data independence* imply DSC, which is strong evidence that restricting attention to DSC will never preclude any real protocol.

Our second area of contribution considers *parameterized* model checking (PMC) of protocols. Here the goal is algorithmic proof over *all* of the infinite configurations of a protocol *family* parameterized by one or more quantities. We develop a technique to automatically abstract a family parameterized by the number of addresses and the number of data values, such that (a subset of) SC of the abstraction implies that of the family. We apply this method successfully to two non-trivial protocols, and suggest user-assisted solutions if the abstraction blows up or is too coarse for successful verification. We also contribute an approach for sound and complete processor-PMC of state assertions. The approach uses BDD-based symbolic model checking, and harnesses the theory of *well-structured transition systems* (WSTS). WSTS disallow *conjunctive guards*, which are found in many protocols. To extend applicability, we provide an automatic reduction for systems with conjunctive guards. Experiments show the efficacy of our conjunctive guard reduction, and that our approach scales better with the local state of each processor when compared with the classical WSTS algorithm.

Contents

Abstract iii

Contents v

List of Tables ix

List of Figures xi

Notations xiii

Acknowledgments xvii

1 Introduction 1

1.1 Memory Model Verification 2

1.2 Parameterized Verification 4

1.3 Contributions 5

1.4 Organization of Thesis 7

2 Decisive Sequential Consistency 9

2.1 Introduction 9

2.2 Preliminary Definitions 10

2.3 Definition of Decisive Sequential Consistency 13

2.3.1 DSC_k 15

2.3.2 Data Independence and DSC 16

2.3.3 Example: Lazy Caching 18

2.4 Complexity of Trace Problems 20

2.5 Complexity of Model Checking DSC$_k$ 26
2.5.1 View Windows 27
2.5.2 VW-Boundedness 35
2.6 Complexity of Model Checking DSC 39
2.6.1 The Protocol $\mathcal{P}_\varphi$ 41
2.6.2 Proof that $\mathcal{P}_\varphi$ is DSC Implies φ 45
2.6.3 Proof that φ Implies $\mathcal{P}_\varphi$ is DSC 51
2.7 Undecidability of Prefix-Closed SC 54
2.7.1 The Protocol $\mathcal{P}_\mathcal{C}$ 56
2.7.2 The Proof 59

3 Address and Value Parameterization 63

3.1 Introduction 63
3.2 Preliminaries 64
3.3 Verification Approach 66
3.4 A Protocol Description Formalism 73
3.4.1 Syntax 74
3.4.2 Semantics 75
3.5 A Candidate Q 76
3.6 Experimental Results 83
3.7 Comments on our Abstraction 86
3.7.1 Comparison to Other Abstractions 87
3.7.2 Example of Verification Failure 89

4 Process Parameterized Verification using BDDs 93

4.1 Introduction 93
4.2 Well-Structured Transition Systems 95
4.2.1 Definitions 95
4.2.2 Examples 96
4.2.3 The Classical Algorithm 100
4.3 Our Approach 102

4.3.1 Nicely Sliceable WSTS 102
4.3.2 Our Algorithm 106
4.3.3 Parametric Initial States 110
4.3.4 An Optimization 113
4.3.5 Near Necessity of the Convergence Theorem 114
4.4 Conjunctive Guard Reduction 116
4.5 Evaluation 121
4.5.1 Implementation 121
4.5.2 Comparison to Standard Approach 123
4.5.3 Petri Nets 124
4.5.4 MESI Protocol 125
4.5.5 German's Protocol 127

5 Related Work **129**
5.1 General Protocol Verification 130
5.2 Memory Model Verification: No Model Checking 130
5.3 Memory Model Verification: Model Checking 132
5.4 Parameterized Verification 136
5.5 Other Related Work 139

6 Conclusions and Future Work **141**
6.1 Decidability of DSC 142
6.2 Hardness of DSC_k 145
6.3 Universal DSC 147
6.4 Extending/Improving Chapter 3 148
6.5 Future Directions for Chapter 4 151

Bibliography **153**

List of Tables

3.1 Results for the Piranha and Wisconsin Directory Protocols 86

4.1 Run of our algorithm against example petri net 111

4.2 Results for selected petri nets 124

4.3 Results for the MESI protocol with PCG over multiple blocks 127

4.4 Results for German's Protocol 127

List of Figures

1.1 Abstract view of a multiprocessor shared memory protocol 3

1.2 A parallel program based on Decker's Algorithm 3

2.1 A coloured trace and its reordering 15

2.2 The architecture of Lazy Caching 19

2.3 Lazy Caching in guarded commands 20

2.4 The trace $\mathsf{ec}(j)$. 23

2.5 VW and VW operations referred to in Examples 1 and 2 31

2.6 The DSC_2 trace used in Example 3 36

2.7 The protocol $\mathcal{P}_\varphi$ 41

2.8 The trace $\mathsf{eva}_b(i)$ 43

2.9 The trace $\mathsf{ec}'(j)$ 44

2.10 The inheritance edges discussed in the proof of Lemma 18 50

2.11 The traces $\mathsf{eva}_\mathsf{F}(i)_0^A$ and $\mathsf{eva}_\mathsf{F}(i)_1^A$ 55

3.1 A non-SC trace that is per-address simple SC 64

3.2 Relationship between the traces mentioned in the proof of Theorem 12 71

3.3 Example of a QFA for which $\perp$-abstraction is too coarse 88

3.4 Example AVPPF $\mathcal{C}$ for which our $\exists$-abstraction is too coarse to verify SSC 90

4.1 The relationship between $\preccurlyeq$ and $\rightarrow$ required by Def. 41 97

4.2 The classical algorithm for the WSTS covering problem 101

4.3 A diagrammatic presentation of Def. 49 104

4.4 Our algorithm for the NSW covering problem 109

4.5 Example petri net 111
4.6 The petri net and upward-closed set discussed in the proof of Theorem 20 115
4.7 The petri net $ME(h)$ 124
4.8 Execution times on the petri net $ME(h)$ 126

6.1 The stripe rearrangement described in the proof of Theorem 23 144

Notations

$\mathbb{N}$	the natural numbers	10
$\mathbb{N}_n$	the set $\{1, \ldots, n\}$	10
$\mathbb{B}$	the set of boolean constants $\{\mathsf{T}, \mathsf{F}\}$	10
$\lvert\tau\rvert$	the length of the string τ	10
ϵ	the empty string/trace	10
Σ^*	the set of all strings over an alphabet Σ	10
$\tau(i)$	the ith element of string τ	10
$\tau[i]$	the prefix of string τ of length i	11
$\tau \upharpoonright A$	the projection of string τ onto the symbols in A	11
$\prod_{i=1}^{k} \sigma_i$	the string concatenation $\sigma_1 \ldots \sigma_k$	11
$\mathcal{M}$	the set of memory actions	11
$op(\alpha)$	the first component of memory action α (either R or W)	11
$proc(\alpha)$	the second component (processor) of memory action α	11
$addr(\alpha)$	the third component (address) of memory action α	11
$val(\alpha)$	the fourth component (data value) of memory action α	11
(R, p, a, v)	a read action	11
(W, p, a, v)	a write action	11
$lw(\tau, i, a)$	the last write to address a not greater than index i in τ	11
$lpa(\tau, p)$	the index of the last action of processor p in τ	12
τ^{π}	the trace obtained by permuting τ with the permutation π	12
$<_{\tau}^{po}$	the processor order on the indices of trace τ	12
$\mapsto_{\tau}^{\pi}$	the inheritance relation on the indices of τ induced by reordering π	12

$lr(\sigma, i)$	the last read relative to index i in serial trace σ	12
$IR(\sigma, j, a)$	the inheritance range predicate	13
$\mathcal{L}(A)$	the language accepted by automaton A	13
$traces(\mathcal{P})$	the set of traces of protocol $\mathcal{P}$	13
$\langle\tau\rangle$	the atomicity construct involving trace τ	21
$\mathsf{eva}(i, b)$	a trace that encodes the assignment of b to variable x_i	23
$\mathsf{ec}(j)$	a trace that encodes clause c_j	23
ps	a position sequence for a serial trace	28
$VW(\sigma, \mathsf{ps})$	the VW-set of serial trace σ and position sequence ps	28
v	a sequence of views	28
lp	a processor position function	28
$\leq_{vw}$	a partial order on view windows	30
$\rightsquigarrow_\tau$	view window τ-evolution relation	32
$\mathsf{evalq}(\varphi)$	the boolean value of QBF formula φ	40
$\mathsf{eva}_b(i)$	encodes the assignment of b to variable x_i and selects some value for y_i	41
CLS	the trace that encodes the clauses of a QBF formula	41
CA	the trace that checks the current variable assignment	41
Σ_n	the alphabet of an n-counter machine	54
$\Sigma_n^\$$	Σ_n augmented with the end of string marker \$	57
$\mathcal{J}$	a finite automaton such that $\mathcal{L}(\mathcal{J})$ equals (2.16)	56
ϕ	a function of the form ${\Sigma_n^\$}^* \rightarrow \mathcal{M}^*$	58
$\mathfrak{F}$	an address and value-parameterized protocol family	64
$\mathsf{scg}(\tau)$	the simple constraint graph of unambiguous trace τ	67
$\models$	the satisfaction relation between m-states and QFAs	76
ξ	the abstract address	76
$\mathsf{sub}(\psi)$	the substitution operation on QFA ψ	77
$\mathsf{sub}_1(\psi)$	a variant of $\mathsf{sub}(\psi)$	77
Q	the abstraction of an AVPPF	77
H	a simulation relation between m-states and abstract states	80
$\perp$	the "no-information" logic value	87

$\preccurlyeq$	a preorder or a well-quasi-ordering	95
$\uparrow Y$	the upward-closure of set Y	95
$\mathsf{basis}(U)$	the unique canonical basis of upward-closed set U	96
$\preccurlyeq_{\mathsf{M}}$	the inclusion ordering on markings	97
${}^{\bullet}t$	the pre-vector of petri net transition t	98
$t^{\bullet}$	the post-vector of petri net transition t	98
$a!\ \ a?$	broadcast protocol rendezvous labels	99
$a!!\ \ a??$	broadcast protocol broadcast labels	99
$\preccurlyeq_{\mathsf{W}}$	the subword ordering	100
$\mathsf{pre}(U)$	the preimage of a set U	100
$\mathsf{pb}(Y)$	a pred-basis of finite set Y	100
$\mathsf{w}(x)$	the weight of a dwqo element x	103
$\mathsf{bw}(U)$	the base weight of an upwards-closed set U	103
$\mathsf{br}(Y, i)$	the backwards reachability from Y not exceeding weight i	107
Lift	the lifting operator	108
$\#(w, \ell)$	the number of occurrences of symbol ℓ in string w	111
Σ_{pcg}	the set of properly conjunctively guarded actions	117

Acknowledgments

I thank my parents for sticking with me through my tumultuous teenage years, which persisted well into my twenties, and for their ongoing support. I thank Claire and Joshua for their tolerance of my temperament as I attempted to balance my new found family life, my irregular sleeping patterns, and pursuit of this degree.

Being co-supervised by Anne Condon and Alan Hu has been a truly gratifying experience. Alan has the uncanny ability to attach intuition to technical concepts, while conversely he often phrases social, political, and everyday scenarios in terms of the mathematical. He has been a great mentor in all of these areas and more. Anne's abilities as a theoretician never cease to amaze me; having her as a supervisor has been invaluable. Anne has also been a source of sage advice for non-technical aspects of my career. Despite having perpetually full plates, both Alan and Anne have always made time for me whenever requested. I also thank Mark Greenstreet (the "other" ISD Lab professor) and Shaz Qadeer (the third member of my supervisory committee), who were both sources of encouragement and insight whenever called upon.

Finally, I thank my brothers and all my good friends who have been exceptional collaborators at running amok.

JESSE BINGHAM

The University of British Columbia
August 2005

Chapter 1

Introduction

Shared memory multiprocessors (SMMP) are becoming a pervasive computing platform, and the importance of these machines will only increase with the current emergence of the chip-multiprocessor [60]. Perhaps the most difficult aspect of designing a SMMP is the hardware-based protocol that facilitates the sharing of memory by multiple processors. Performance is paramount in such protocols, hence they are typically highly optimized. This, coupled with the copious concurrency inherent in these systems, makes them extremely bug-prone and hence a natural target for *formal verification*. Formal verification strives to find bugs and/or certify correctness by constructing a mathematically rigorous, i.e. *formal*, proof that a model of a system adheres to its specification. This thesis is about formal verification of SMMP memory protocols.

Since the seminal work in the late 1960s, the majority of the formal verification literature has focused on employing computers to reduce the level of human expertise and time needed to construct proofs. Here, the computer plays two (not necessarily distinct) roles: to *ensure soundness* of the proof, and to *automate* aspects of the proof. A popular method that achieves both of these goals is called *model checking*, which, for the purposed of this thesis, simply refers to any totally algorithmic means of answering a verification query about a system.[1]

We treat two orthogonal goals surrounding SMMP memory protocol (hereafter simply *protocol*) verification: *memory model verification* and *parameterized verification*. Both of these are ambitious for contrary reasons; the former tackles verification of a difficult *specification* while the latter tack-

[1]In some contexts, *model checking* implies systematic state space exploration of a finite-state system; however when discussing complexities of model checking problems and parameterized verification (as we do) our broader sense is appropriate.

les verification of a difficult *system*. In the ensuing Sects. 1.1 and 1.2, we respectively discuss these goals. The contributions of the thesis are outlined in Sect. 1.3, and finally a brief roadmap is presented in Sect. 1.4.

1.1 Memory Model Verification

The memory model [4] is the ultimate specification of the protocol, and has been aptly described as the contract between the multiprocessor designer and its programmer [113]. Consider the abstract view of an SMMP memory protocol depicted in Fig. 1.1. Here we see n clients, the processors, connected to the protocol. The protocol typically involves common hardware features such as caches, memories, FIFO queues, networks, control logic, and buses. As the clients, the processors instigate *memory events* across the conceptual interface between the processors and the protocol. For the purposes of this thesis, memory events will comprise *write* events, wherein a processor writes a *data value* to a specified *address*, and *read* events, wherein a processor reads a data value from an address.[2]

A memory model can be viewed as a set of strings of these events specifying the legal behaviors at the interface. A given protocol satisfies the memory model if all its behaviors are permitted by the memory model. The memory model considered in this thesis is called *Sequential Consistency* (SC), which was first articulated by Lamport [88] and more recently promoted by Hill [77]. The intuition behind this model is this: on a sequentially consistent SMMP, memory *appears* to service requests atomically. Usually, due to various optimizations, most notably *caching*, this appearance is illusory.

The following example should illuminate SC and memory models in general. Consider the parallel program of Fig. 1.2, which strives to allow at most one of the processors p_1 and p_2 to execute their respective critical sections, and assumes that initially $\texttt{flag1} = \texttt{flag2} = 0$. Suppose that a run of this program yields the following sequence of memory events on the interface:

$$(W, p_1, \texttt{flag1}, 1)(W, p_2, \texttt{flag2}, 1)(R, p_1, \texttt{flag2}, 0)(R, p_2, \texttt{flag1}, b) \tag{1.1}$$

Here, (W, p, a, d) (resp. (R, p, a, d)) is the memory event in which processor p writes (resp. reads) value d to (resp. from) address a. Note that (1.1) actually defines two different sequences, depending

[2] The third type of memory event, which are not considered in this thesis, are explicit synchronization commands, which are supported by many multiprocessors.

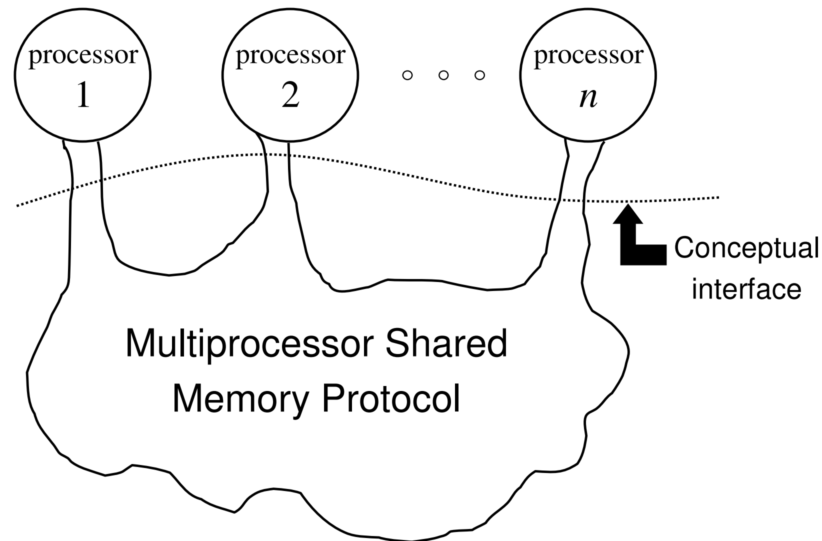


Figure 1.1: Abstract view of a multiprocessor shared memory protocol

Processor p_1

```
flag1 := 1;
if (!flag2) {
    critical_section();
}
```

Processor p_2

```
flag2 := 1;
if (!flag1) {
    critical_section();
}
```

Figure 1.2: A parallel program based on a fragment of Decker's Algorithm [15]. Assuming that initially $\texttt{flag1} = \texttt{flag2} = 0$, when executed on a sequentially consistent multiprocessor the code will allow at most one of the two processors to enter its critical section.

on whether the final event reads $b = 0$ or $b = 1$. Regardless of b, the reader might find (1.1) to be aberrant; the second event writes 1 to address `flag2`, but then later the value 0 is read from the same address. On a uniprocessor such behavior would be indicative of a broken memory system. However, in the multiprocessor setting, SC allows such behavior. Indeed, (1.1) with $b = 1$ satisfies SC, and clearly this behavior only allows p_1 into its critical section.

Assuming an SC machine, one can easily prove that the program of Fig. 1.2 will never allow both processors into their critical sections. This is not necessarily the case for more permissive (a.k.a *weaker*) memory models; several models would allow (1.1) with $b = 0$, hence mutual exclusion would be violated.

As seen in the above example, the conformance of the protocol to the intended memory model is crucial; violation of this "contract" can have disastrous consequences on the behavior of parallel programs. Hence, it is desirable to formally prove correctness with respect to the memory model, or at least harness formal verification to weed out bugs and thus increase confidence in correctness. In this thesis, our attention is focused on model checking a protocol to determine memory model adherence, and the memory model we consider is sequential consistency. Though many real machines are not sequentially consistent, the other prevalent models are often defined as relaxations of SC, and hence we believe that research on verification of SC can inform techniques for other models.

1.2 Parameterized Verification

In theory, a real protocol implemented in hardware has only a finite number of state holding elements, and thus only a finite number of reachable states. However, the number of reachable states is much greater than the number of SAPs in the universe, where a *SAP* is the reader's favorite sub-atomic particle. Hence, in practice, real protocols should be regarded as infinite state systems.

Related is the notion of *parameterization*; typical protocol descriptions are meant to work for any number of processors, shared addresses, data values per address, and other parameters such as cache sizes, queue depths, network capacities, etc. Ideally, we would prove correctness of all members of this infinite *family* of related protocols. Of course model checking generally falls substantially short of this goal, since only very small instantiations (i.e. 2 processors, 2 addresses, 2 value per address) can be handled. Correctness of such miniscule configurations builds confidence that the entire family is correct, but special techniques are required if we wish to make such generalizations logically sound.

Parameterized Model Checking (PMC) refers to methods that extend model checking for certain classes of systems so that correctness can be proven for all parameter values. Such methods roughly fall into two classes: abstraction-based approaches which are *sound* but typically *incomplete*,[3] and those methods that are both sound and complete. This thesis explores an example of each of these. We note that since PMC in a sense subsumes finite state model checking, it is tricky business. Typically, both the class of properties and the class of systems that can be handled by PMC techniques

[3]Soundness means that the method will never pass an erroneous system. Conversely, completeness means that the method will never fail a correct system.

are restrictive. Combining PMC with memory model verification (as we will do in Chapter 3) is indeed an important and formidable objective.

1.3 Contributions

Our contributions are divided amongst the three main thesis chapters. Chapter 2 treats the complexities of several verification problems related to a new subset of SC called DSC. Chapter 3 meshes the two goals of memory model verification and PMC into an abstraction-based method that verifies a subset of SC for arbitrary numbers of addresses and data values per address. Finally, Chapter 4 focuses on PMC across processor counts, which is the hardest dimension to tackle. Here only state assertions are verified, as opposed the more difficult property of memory model adherence. We now elaborate on these three chapters.

Chapter 2. Here we study the problem of model checking a finite state (i.e. unparameterized) protocol for adherence to sequential consistency. It turns out that for even a finite state protocol this is a computationally hard problem; unlike model checking for temporal logics such as CTL or LTL, which have algorithms that are polynomial in the size of the state space of the protocol, SC is undecidable [8]. Full SC allows for pathological behaviors in which the protocol *must necessarily* revoke decisions regarding which write event bestowed the value imparted to a given read event. In hopes of alleviating complexity yet still capturing the behaviors of any conceivable protocol, we define a restricted version of SC called *decisive SC* (DSC). DSC forbids precisely those pathological behaviors mentioned above. We also define a variant of DSC called DSC_k (for any integer $k \geq 1$), which in some sense bounds the historical information required to maintain DSC. Our key results regarding the complexities of related verification problems are:

- SC in conjunction with the ubiquitous property of *data independence* imply DSC, which is strong evidence that restricting attention to DSC will never preclude any real protocols.

- Verifying either full DSC or DSC_k for $k \geq 2$ of a single behavior is NP-complete.

- Model checking a bounded variant DSC_k can be accomplished in EXPSPACE.

- Model checking DSC is PSPACE-hard.

- Model checking SC itself remains undecidable when we require the set of behaviors of a protocol to be prefix-closed; this was *not* required by Alur et al. [8] in their proof that model checking SC is undecidable.

The foremost problem left open by Chapter 2 is the decidability of model checking DSC. Note that our PSPACE-hardness result is exactly that, only a proof that the problem is at least as hard as anything in PSPACE, and says nothing about decidability. As the concentration is complexity and computability results, Chapter 2 is the most theoretical of the thesis. The following Chapter 3 actually involves experiments with real protocols.

Chapter 3. Here we develop a method for PMC over the address and data value parameters; the property verified is a restricted version of SC called *simple SC*. The method uses abstraction; we automatically construct a finite state protocol that abstracts the entire parameterized family of protocols in such a way that simple SC of the abstraction implies that of the whole parameterized family. We apply the technique successfully to two nontrivial example protocols. In principle, the technique can be carried out automatically, however in practice, we find that it can fail for two different reasons: the state space of the abstraction blows up, or the abstraction is too coarse. We spell out means of remedying both of these situations that require only a modest level of user assistance. Later, in Sect. 6.4 we suggest how one might extend the method of Chapter 3 to handle non-simple SC protocols.

Chapter 4. Due to their strong interactions, parameterizing over the number of processors is the most difficult dimension to handle in PMC. Our final contribution is a technique that can perform automatic, sound and complete processor-parameterized PMC; the property verified here is the unreachability of a set of error states. The approach uses BDD-based symbolic model checking, and harnesses the theory of *well-structured transition systems* (WSTS), which is a class of infinite state systems that enjoy decidability of certain verification problems. The technique involves analysing the system restricted to 1 processor, then 2, then 3, etc., until either an error is found, or a certain convergence condition is reached. The latter case allows one to conclude correctness for an arbitrary number of processors. This convergence condition and how it is computed using BDDs are the main theoretical contributions of the chapter.

Well-structuredness precludes several attributes typically found in protocols; one of these is the *conjunctive guard*. To extend the reach of our approach, we provide a sound and automatic reduction

for systems with conjunctive guards. This allows us to perform PMC of German's Protocol (a well-known challenge problem for PMC) in about one minute of processing time. Several experiments are presented that demonstrate how our approach scales much more gracefully with the size of the *local state* of each processor when compared with the classical algorithm from the WSTS literature.

1.4 Organization of Thesis

Following this introductory chapter are the three core chapters summarized in the previous section. Subsequently, Chapter 5 highlights related work, and Chapter 6 concludes the thesis and describes future research directions. Sect. 2.2 defines much of the notation used throughout the thesis, other notation is introduced on a need-to-know basis.

Chapter 2

Decisive Sequential Consistency

2.1 Introduction

This section is devoted to motivating, defining, and exploring the verification complexity of a multiprocessor correctness criterion we call *decisive sequential consistency* (DSC). DSC amends the well-known memory model *sequential consistency* by precluding certain behaviors that are both difficult to reason about and unlikely to exist in a real shared memory system.

Intuitively, a shared memory system is sequentially consistent (SC) if it *appears*, from the point of view of the client processors, that the system services memory requests atomically. In other words, the system must be indistinguishable, from the processors' perspective, from a *serial memory*, in which each read inherits the value written by the (temporally) last write to the same address. An outside observer, i.e. one that can see the histories of all processors, could notice behaviors that are not serial, however, any behavior of the SC system must be *explainable* as a behavior of serial memory. Formally, the *explanation* will be a *reordering* (a.k.a. *shuffling*, *permutation*) of the original behavior.

One can define SC protocols that behave very strangely. We feel that the most pathological behavior admitted by SC is the ability for the explanation to radically change as history unfolds. This notion of disallowing radical changes is formally defined in Sect. 2.3, wherein we show that this notion is equivalent to the requirement that, in the explanation, no read inherits its value from a write that occurs in its temporal future. Any protocol that requires this "prophetic inheritance" must either be forcing certain necessary writes to occur, or has been modelled with write and read events associated with the incorrect protocol transitions. Hence, we introduce DSC as a correctness

criterion that rules-out such peculiarities. Informally, DSC requires that when a memory read event occurs, the protocol "knows" which write event the data value stemmed from, and the protocol never needs to "change its mind" in the future in order to maintain SC. All SC protocols with which we are familiar actually implement the slightly stronger property DSC.[1]

This chapter makes the following contributions. Sect. 2.2 formalizes the notions of SC and shared memory protocols. Sect. 2.3 motivates and defines our notion of *decisive SC* (DSC). and also defines a version of DSC that is bounded in some sense. We provide evidence that DSC is the "right" notion by showing that an oft-studied SC protocol has this property, and further that *any* data independent SC protocol must be DSC. Sects. 2.4, 2.5, and 2.6 respectively consider the complexities of testing a single protocol behavior for (bounded) DSC, model checking a protocol for the bounded version of DSC, and model checking a protocol for unbounded DSC. We find that these three problems are respectively in the classes NP-complete, EXPSPACE, and PSPACE-hard. Note that this final result is merely a lower-bound;[2] the decidability of model checking DSC is an open problem.

We conclude the chapter with Sect. 2.7, wherein we prove that model checking SC for our notion of *protocol*, which requires the set of behaviors to be prefix-closed, is undecidable. This refines an earlier result of Alur et al. [8] stating that SC is undecidable for protocols that are not (necessarily) prefix-closed.

2.2 Preliminary Definitions

Let $\mathbb{N}$ denote the set of natural numbers $\{0, 1, 2, \ldots\}$. For a natural n, let $\mathbb{N}_n = \{1, \ldots, n\}$. The boolean constants for true and false will respectively be denoted T and F, and we define $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$.

Strings. Given an alphabet Σ, a *string* or *word* over Σ is a mapping $\tau : \mathbb{N}_\ell \to \Sigma$ for some ℓ, and we define the length of τ to be $|\tau| = \ell$. We sometimes refer to $i \in \mathbb{N}_\ell$ as an *index* (of τ). The *empty string*, denoted ϵ, is the unique string of length 0. The set of all strings over an alphabet Σ is denoted Σ^*. A string can be expressed by writing its symbols in order, i.e. $\tau(1) \ldots \tau(|\tau|)$. Hence

[1]Lamport, who originated the notion of sequential consistency [88], wrote a footnote on page 195 of his book [89] that is worthy of duplication here: "The freedom to change explanations, which a sequentially consistent memory allows, could conceivably be used to permit a more efficient implementation, but it's not easy to see how."

[2]But a hard-won lower bound!

the use of the terms *left* and *right* to refer to features of the string makes sense, we sometimes use the term *last* to mean rightmost. The prefix of τ of length i is denoted $\tau[i]$. Given a string τ and a set A, the *projection* of τ onto A is the string $\tau \uparrow A$ obtained by deleting all symbols not in A. For $\alpha \in \Sigma$ we say that a string $\tau' \in \Sigma^*$ is obtained from τ by *inserting* α *at position* j if $\tau' = \tau(1)\dots\tau(j)\alpha\tau(j+1)\dots\tau(|\tau|)$, where j may be any element of $\mathbb{N}_{|\tau|} \cup \{0\}$.

We will use the term *substring* to refer to a contiguous substring. A string may be expressed as the concatenation of symbols and/or other strings; we simply juxtapose strings and symbols to denote the larger string formed by their concatenation. Also, given strings $\sigma_1, \dots, \sigma_k$, we will use

$$\prod_{i=1}^{k} \sigma_i$$

to denote the concatenation $\sigma_1 \dots \sigma_k$.

Mathematically, there is of course a distinction between a string σ and an occurrence of σ as a substring of some other string. We have not introduced a formal notation to make this distinction, the intention will either be clear from context or the word *occurrence* will be used to disambiguate.

Memory actions and traces. Given sets P, A, and V, the set of *memory actions* is the set $\mathcal{M}(P, A, V) = \{R, W\} \times P \times A \times V$. Intuitively, P, A, and V are respectively the processors, addresses, and data values involved in a shared memory protocol. A memory action will typically be called simply an *action* or sometimes an *event*. When P, A, and V are irrelevant or clear from context, we simply refer to the set of memory actions by $\mathcal{M}$. Unless otherwise noted, P, A, and V will be sets of the form $\mathbb{N}_n$, $\mathbb{N}_m$, and $\mathbb{N}_v$ for some n, m, and v, respectively. For $\alpha = (o, p, a, v) \in \mathcal{M}$, we define $op(\alpha) = o$, $proc(\alpha) = p$, $addr(\alpha) = a$, and $val(\alpha) = v$. We call a string over $\mathcal{M}$ a *trace*. We also define subsets of $\mathcal{M}$ using the wild-card symbol $*$. For instance, $(*, p, a, *)$ is the set $\{\alpha \in \mathcal{M} \mid proc(\alpha) = p \wedge addr(\alpha) = a\}$. Sometimes we also use subsets in various components, for example $(*, \{1, 2\}, a, *)$ is the set of all actions α such that $proc(\alpha) \in \{1, 2\}$ and $addr(\alpha) = a$. The set $(R, *, *, *)$ constitutes the *read actions* while the elements of $(W, *, *, *)$ are the *write actions*.

Given a memory trace τ, an index i, and address a, we define $lw(\tau, i, a)$ to be the greatest index $j \leq i$ such that $\tau(j) \in (W, *, a, *)$ or to be 0 if no such j exists. Intuitively, $lw(\tau, i, a)$ is the index of the last write to address a that is not greater than i. For address a and value v, we define a v-*phase of* a to be an interval $[i, j]$ such that $\tau(i) \in (W, *, a, v)$, $lw(\tau, i', a) = i$ for all $i' \in [i, j]$, and either

$\tau(j+1) \in (W,*,a,*)$ or $j = |\tau|$. Let τ' be an occurrence of a substring in τ, assumed not to contain any writes to a. We say that τ' is *in a v-phase of a* if there exists a v-phase of a that contains all indices of τ', i.e. if the last write to a in τ prior to τ' wrote value v. For processor p, the index of the last action of τ in $(*,p,*,*)$ is extracted via $lpa(\tau,p) = \max(\{i \mid proc(\tau(i)) = p\} \cup \{0\})$. Here, the acronym lpa stands for *last processor action*.

A trace σ is said to be *serial* if, for all indices i such that $op(\sigma(i)) = R$ we have that i is in a $val(\sigma(i))$-phase of $addr(\sigma(i))$. In other words, in a serial trace each read action returns the value of the last write action to the same address.[3] As a notational convenience, we define $val(\sigma(0))$ to be a distinguished constant $\perp$, which signifies that no value has been assigned. We typically use σ to represent a serial trace, while τ represents a trace that is not necessarily serial.

A *reordering* of a trace τ is a permutation π on $\mathbb{N}_{|\tau|}$, and we define the trace τ^π by

$$\tau(\pi^{-1}(1)) \dots \tau(\pi^{-1}(|\tau|))$$

In a slight abuse of terminology, we will sometimes also refer to τ^π as a *reordering*. We allow the application of a reordering π to a prefix of length $k < |\tau|$ by defining the permutation $\pi' : \mathbb{N}_k \to \mathbb{N}_k$ such that $\pi'(i) < \pi'(j) \Leftrightarrow \pi(i) < \pi(j)$ for all $i,j \in \mathbb{N}_k$, and then defining the reordered prefix $\tau[k]^\pi$ to be equal to $\tau[k]^{\pi'}$. Associated with a trace is a partial order which places actions of the same processor in the order they occur in the trace; the resulting relation is called the *processor order*. Formally, given a trace τ we define the partial order $<^{po}_\tau$ on the indices such that $i <^{po}_\tau j \Leftrightarrow (proc(i) = proc(j)) \wedge (i < j)$. A *serial reordering* of τ is a reordering π such that τ^π is serial, and further π respects $<^{po}_\tau$, i.e. $i <^{po}_\tau j$ iff $\pi(i) < \pi(j)$, or equivalently, for all processors p, we have $\tau \uparrow (*,p,*,*) = \tau^\pi \uparrow (*,p,*,*)$. A trace is said to be *sequentially consistent (SC)* if it has a serial reordering. If τ is SC and has serial reordering π, then we define an *inheritance relation* $\mapsto^\pi_\tau$ on the indices of τ by $i \mapsto^\pi_\tau j$ if $op(\tau(i)) = W$, $op(\tau(j)) = R$, $addr(\tau(i)) = addr(\tau(j))$, and $\pi(i) = lw(\tau^\pi, \pi(j), addr(\tau(j)))$. When $i \mapsto^\pi_\tau j$ we say that j *inherits from* i. If a trace σ is serial, then we use $\mapsto_\sigma$ to denote $\mapsto^{\mathrm{id}}_\sigma$, where id is the identity permutation.

Let σ be a serial trace. For any i such that $op(\sigma(i)) = W$, we define the *last read of* i by $lr(\sigma,i) = \max(\{i' \mid i \mapsto_\sigma i'\} \cup \{i\})$. Intuitively, $lr(\sigma,i)$ provides the *last read* that inherits from i.

[3]For simplicity's sake, we have chosen to disallow initial values; this approach is also taken by Gibbons and Korach [66]. It should be clear that all theory herein can be generalized to handle formalizations that allow reading of initial data values.

For any $j \in \{0, \ldots, |\sigma|\}$ and address a, we define the *inheritance range* predicate $IR(\sigma, j, a)$ that is true iff $lw(\sigma, j, a) \neq 0$ and $j < lr(\sigma, lw(\sigma, j, a))$. Thus $IR(\sigma, j, a)$ holds whenever position j is between a write to address a and a read that inherits from the write.

Automata and Protocols. An *automaton* is a quintuple $A = (S, \Sigma, \delta, s_0, F)$, where S is a set of *states*, Σ is a finite *alphabet*, $\delta \subseteq S \times \Sigma \times S$ is the *transition relation*, $s_0 \in S$ is the *initial state*, and $F \subseteq S$ are the *accepting states*. A is said to be a *finite automaton* if S is finite. Let δ^* be the natural generalization of δ to strings; formally we define $\delta^* \subseteq S \times \Sigma^* \times S$ as the smallest relation such that

- $(s, \epsilon, s) \in \delta^*$ for all $s \in S$, and
- whenever $(s, w, s'') \in \delta^*$ and $(s'', \alpha, s') \in \delta$, we have $(s, w\alpha, s') \in \delta^*$.

The *language* of A is the set of strings $\mathcal{L}(A) \subseteq \Sigma^*$ such that $w \in \mathcal{L}(A)$ if and only if there exists $s_f \in F$ such that $(s_0, w, s_f) \in \delta^*$. The *size* of A, denoted $|A|$, is the number of bits needed to encode A, which is roughly proportional to $|\delta|$.

A *shared memory protocol* (or simply *protocol*) $\mathcal{P}$ is a finite automaton in which all states are accepting and the alphabet is a set of memory actions along with a disjoint set of *silent* (or *internal* actions) I. Hence the final component of an automaton is redundant when specifying protocols, and we express $\mathcal{P}$ as a quadruple (S, Σ, δ, s), where $\mathcal{M}(P, A, V) \subseteq \Sigma$ for some P, A, and V and $\Sigma \setminus \mathcal{M}(P, A, V)$ contains no memory actions. We say that $\mathcal{P}$ *involves* $|P|$ processors, $|A|$ addresses, and $|V|$ data values. The *traces* of $\mathcal{P}$ are defined to be the projection of its language onto memory actions, i.e.

$$\mathit{traces}(\mathcal{P}) \ = \ \{w \!\uparrow\! \mathcal{M} \mid w \in \mathcal{L}(\mathcal{P})\}$$

Any property of traces is automatically extended to refer to a protocol if all its traces satisfy the property, for example a protocol is sequentially consistent if all its traces are sequentially consistent.

2.3 Definition of Decisive Sequential Consistency

In this section, we define two SC variants, DSC and PTSC, and we prove their equivalence. Then, for each $k \in \mathbb{N}$, we define a class DSC_k, which is a subset of DSC that is bounded in some sense.

Definition 1 (Decisive SC). *A trace τ is said to be* decisive sequentially consistent (DSC) *if there exists a serial reordering π of τ such that for any prefix $\tau[\ell]$ and $1 \leq i, j \leq \ell$ we have: (1) $\tau[\ell]^\pi$ is serial, and (2) $i \mapsto^\pi_{\tau[\ell]} j \Leftrightarrow i \mapsto^\pi_\tau j$. In this case we say that π is a* DSC-reordering *of τ.*

On an intuitive level, a DSC protocol is a SC protocol that never needs to "change its mind" regarding which write a given read inherits from. To our knowledge, the only place in the literature where this characteristic is paid heed is Lamport's book [89, Chapter 11.2]. There, Lamport calls a notion similar to DSC *serial memory* (an unfortunate terminological conflict with this thesis). However, his emphasis is on specification rather than verification.

A trace being DSC implies that all prefixes of the trace are SC (moreover they are all DSC). However, having all prefixes SC is insufficient for DSC, as this example trace ρ demonstrates.[4] To aid comprehension, we attach a number underneath each memory action giving its position in ρ.

$$\rho = \underset{1}{(W, p_2, a_1, 2)}\underset{2}{(W, p_2, a_1, 1)}\underset{3}{(W, p_2, a_2, 1)}\underset{4}{(R, p_1, a_2, 1)}\underset{5}{(W, p_3, a_2, 1)}\underset{6}{(R, p_1, a_1, 2)}$$

A serial reordering π exists:

$$\rho^\pi = \underset{5}{(W, p_3, a_2, 1)}\underset{4}{(R, p_1, a_2, 1)}\underset{1}{(W, p_2, a_1, 2)}\underset{6}{(R, p_1, a_1, 2)}\underset{2}{(W, p_2, a_1, 1)}\underset{3}{(W, p_2, a_2, 1)}$$

It can be shown that *any* serial reordering must have 4 inherit from 5; this follows from the fact that any serial reordering must order 6 prior to 2. However, restricting such a reordering to the prefix τ of length 4 will not give a serial reordering. Continuing our example, we find

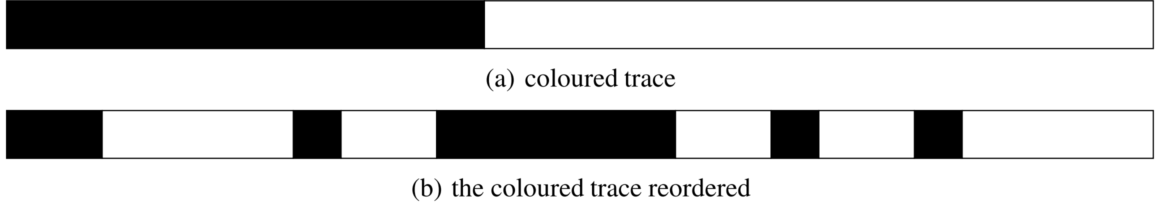
$$\rho[4]^\pi = \underset{4}{(R, p_1, a_2, 1)}\underset{1}{(W, p_2, a_1, 2)}\underset{2}{(W, p_2, a_1, 1)}\underset{3}{(W, p_2, a_2, 1)}$$

which is clearly not serial. The reader may confirm that all prefixes of τ are SC, for instance $\rho[4]$ may be serially reordered as

$$\underset{1}{(W, p_2, a_1, 2)}\underset{2}{(W, p_2, a_1, 1)}\underset{3}{(W, p_2, a_2, 1)}\underset{4}{(R, p_1, a_2, 1)}$$

We now present a seemingly different SC variant, PTSC, and prove that it is equivalent to DSC. Our motivation for presenting DSC in the guise of PTSC is to provide greater intuition regarding what type of behaviors these models allow and disallow.

[4] Lamport provides a very similar trace on page 187 of his book [89].



(a) coloured trace

(b) the coloured trace reordered

Figure 2.1: A trace with a prefix coloured black and the remainder coloured white, and the striping effect induced on a reordering of the trace.

Definition 2 (past-time SC). *A trace τ is said to be* past-time sequentially consistent (PTSC) *if there exists a serial reordering π of τ such that $i \mapsto_{\tau}^{\pi} j$ implies $i < j$. In this case we say that π is a* PTSC-reordering *of τ.*

Theorem 1. *π is a DSC-reordering of τ if and only if π is a PTSC-reordering of τ.*

Proof. ($\Rightarrow$) Suppose there exist i and j such that $i \mapsto_{\tau}^{\pi} j$ and $i > j$. Consider the prefix $\tau[i-1]$. Clearly $i \mapsto_{\tau[i-1]}^{\pi} j$ cannot hold, since i isn't an index of $\tau[i-1]$. ($\Leftarrow$) Now assume π is a PTSC-reordering of τ. We prove by (decreasing) induction on ℓ that properties 1 and 2 from definition 1 hold w.r.t $\tau[\ell]$. For $\ell = |\tau|$ we have $\tau[\ell] = \tau$, hence the properties hold trivially. Now assume that the properties hold in $\tau[\ell]$. If $\tau(\ell)$ is a write then there does not exist i such that $\ell \mapsto_{\tau[\ell]}^{\pi} i$, thus $\tau[\ell-1]^{\pi}$ satisfies both properties. Now if $\tau(\ell)$ is a read, then $\tau[\ell-1]^{\pi}$ is trivially serial, since removing a read from a serial trace will always yield a serial trace; property 2 is also satisfied given the inductive hypothesis. $\square$

2.3.1 DSC$_k$

Consider a trace $\tau = \tau_1\tau_2$ with DSC-reordering π, and imagine that we colour the events of τ_1 and τ_2 black and white, respectively. From afar, τ would appear as a single black segment (τ_1) followed by a single white segment (τ_2) as in Fig. 2.1(a). Observing τ^{π} from a distance, however, one would see multiple black and white stripes as a result of the shuffling of black and white events. Fig. 2.1(b) depicts how this effect might appear. The striping, in some sense, captures the amount of shuffling between τ_1 and τ_2. We formalize this notion as follows.

Definition 3 (shuffle degree). *Let π be a DSC-reordering of trace τ, and let $\ell \in \{1, \ldots, |\tau|\}$. Let $c_\ell : \{1, \ldots, |\tau|\} \to \{black, white\}$ be defined so that $c_\ell(i) = black$ iff $\pi^{-1}(i) \leq \ell$. Then the* shuffle

degree *of* (τ, π, ℓ) *is equal to*

$$|\{i \in \{0, \ldots, |\tau|\} \mid (i = 0 \vee c_\ell(i) = \mathit{black}) \wedge (c_\ell(i+1) = \mathit{white} \vee i = |\tau|)\}| \tag{2.1}$$

Intuitively, ℓ gives the length of a prefix τ_1 of τ which we colour black, we colour the remainder of τ white, and the shuffle degree is the resulting number of black stripes in the reordering τ^π, with the exception that we add 1 if the leftmost stripe is white.

Definition 4 (DSC-bound,DSC$_k$). *Let π be a DSC-reordering of trace τ, and let k be a positive integer. Then (τ, π) is said to have* DSC-bound k *if for all $\ell \in \{1, \ldots, |\tau|\}$, the shuffle degree of (τ, π, ℓ) is not greater than k. The set DSC$_k$ is defined to be the set of all traces τ having some DSC-reordering π such that (τ, π) has DSC-bound k.*

So a DSC trace τ has DSC-bound k if there exists some DSC-reordering π such that no matter how we factor $\tau = \tau_1\tau_2$ and perform our colouring exercise, we never see more than k black stripes in the reordered trace τ^π. Note that the DSC bound *does not* restrict how far out of sync (i.e. with respect to a reordering) two processors can be. It is the *number* of stripes in the shuffling that is bounded, *not* the number of events in each stripe. This contrasts with other proposed restrictions on SC [74, 117].

All conceivable DSC protocols are DSC$_k$ for some finite k.[5] One can think of k as being the (maximum) number of points in logical time that the protocol maintains information about for future use by processors. Typically this information is implicit in protocol messages, message ordering constraints, and other features of the protocol state, and k depends on the message capacities of various communication resources. For example, in Sect. 2.4 we will see that seriality and DSC$_1$ are equivalent. Hence serial protocols are DSC$_1$, which agrees with the fact that in such protocols all processors always exist at the same point in logical time. Later, in Sect. 2.3.3, we give a concrete example of a DSC$_k$ protocol (with $k > 1$) and how the value of k depends on its architecture.

2.3.2 Data Independence and DSC

An interesting question pertains to the space of protocols that are SC but not DSC: are these of any conceivable practical importance? In this section we answer this question negatively.

[5]This is formalized by Conjecture 1 on page 39.

Data independence (DI) is a property that any real shared memory protocol possesses. Intuitively, DI in a system (of any kind) means that variables of a certain type can only be nondeterministically assigned, copied, and outputted [124].[6] When the system is a protocol, and the type is data values, DI requires that the protocol simply moves data items around, oblivious to the actual data values being moved. Not only is DI inherent in real protocols, it can be (and typically is) enforced by syntactic constraints on protocol descriptions [113, 124].

Here we prove that DI precludes traces that are SC and not DSC. The basic idea behind our proof is very simple. Call a trace *unambiguous* if it has the feature that no two writes to the same address write the same value. For any non-DSC protocol trace τ, DI guarantees the existence of an unambiguous trace τ' with the same structure as τ. This notion of "same structure", defined below via renaming functions, along with the fact that τ is not DSC, imply that τ' is also not DSC. It follows that τ' must have a read event (R, p, a, v) that occurs before the unique event of $(W, *, a, v)$. We then note that an appropriately selected prefix of τ' contains (R, p, a, v) but does not contain any event of $(W, *, a, v)$, thus immediately ruling out the possibility that this prefix is SC.

To formally define DI, it is convenient to do so for a countably infinite set of related protocols, each involving a different set of data values. Our definitions roughly follow Qadeer [113].

Definition 5 (value-parameterized protocol family). *A* value-parameterized protocol family *is a mapping $\mathcal{P}$ that takes each $v \geq 1$ to a protocol $\mathcal{P}(v)$ involving v data values.*

Renaming functions are used to formally define a connection between the traces of the constituent protocols of a value-parameterized protocol family. They take an address and a data value, and return another data value, thusly *renaming* the address's data.

Definition 6 (renaming function). *Given m, v, and v' such that $m \geq 1$ and $1 \leq v \leq v'$, we call a function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \rightarrow \mathbb{N}_v$ a* renaming function*.*

We extend the renaming function λ to a function $\mathcal{M} \rightarrow \mathcal{M}$ by decreeing $\lambda((o, p, j, d)) = (o, p, j, \lambda(j, d))$. We further extend λ to a function $\mathcal{M}^* \rightarrow \mathcal{M}^*$ by applying λ to each event in the trace. We are now equipped to define data independence.

[6] Some literature also allows for equality tests on the type; we do not.

Definition 7 (data independence). *A value-parameterized protocol family $\mathcal{P}$ is said to be* data independent *if for all $v \geq 1$, for any trace τ of $\mathcal{P}(v)$, there exists $v' \geq v$, a renaming function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \rightarrow \mathbb{N}_v$, and an unambiguous trace τ' of $\mathcal{P}(v')$ such that $\lambda(\tau') = \tau$.*

Theorem 2. *Let $\mathcal{P}$ be a data independent value-parameterized family of protocols. Then $\mathcal{P}$ is SC if and only if $\mathcal{P}$ is DSC.*

Proof. ($\Leftarrow$) Trivial ($\Rightarrow$) Suppose, on the contrary, there exists a value-parameterized family $\mathcal{P}$ that is SC and DI, but not DSC. Then there exists $v \geq 1$ and a trace τ of $\mathcal{P}(v)$ such that τ is SC but not DSC. By data independence, there exists $v' \geq v$, an unambiguous trace τ' of $\mathcal{P}(v')$, and a renaming function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \rightarrow \mathbb{N}_v$ such that $\lambda(\tau') = \tau$. It follows that τ' is not DSC, since applying λ to a DSC-reordering of τ' would yield a DSC-reordering of τ. τ' must have a serial reordering π, and, from Theorem 1, π is not a PTSC reordering. Hence there exists $i, j \in \{1, \ldots, |\tau'|\}$ such that $i \mapsto^{\pi}_{\tau'} j$ and $i > j$. Let $(W, p, a, d) = \tau'(i)$. Since τ' is unambiguous, there exists no $i' \neq i$ such that $\tau'(i')$ is a write of value d to address a. Consider the prefix $\tau'[j]$; this string has no such write, yet it does contain $\tau'(j)$ which is a read of d from address a. Therefore, $\tau'[j]$ cannot be SC, which contradicts $\mathcal{P}$ being SC, since it is a trace of $\mathcal{P}(v')$. □

2.3.3 Example: Lazy Caching

As an example of the expressiveness of DSC we show that the well-studied SC protocol Lazy Caching [6] is DSC. Our description of Lazy Caching follows later papers [11,99].[7] The architecture of Lazy Caching is shown in Fig. 2.2. The processors $p_1, \ldots, p_n$ are of course the clients of the protocol and not actually *part* of the protocol. Each processor p_i has its own private cache $cache_i$; the memory events of p_i can be thought of as occurring on the interface between p_i and $cache_i$. Each $cache_i$ communicates with main memory Mem via FIFO queues out_i and in_i. The former of these queues buffers messages en-route from $cache_i$ to Mem, while the latter buffers messages traveling in the opposite direction.

Let P, A, and V be the set of processors, addresses, and data values. For each $(i, j, d) \in P \times A \times V$, Lazy Caching exhibits the familiar memory actions (R, i, j, d) and (W, i, j, d), as well

[7] The primary difference between the description of the original Lazy Caching paper [6] and the later papers [11, 99] is that in the former, the externally visible memory events are split into requests and returns, and hence SC is also defined over this alphabet. In the latter, the visible events are from our $\mathcal{M}$.

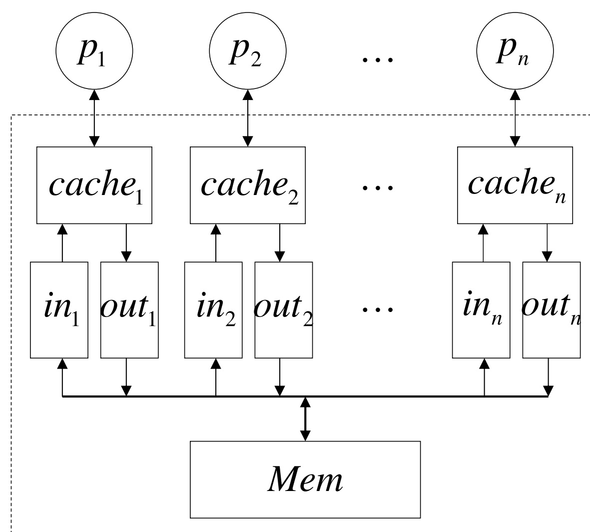


Figure 2.2: The architecture of Lazy Caching

as the following four internal events.

- MemoryWrite$_i(a, d)$. A message is dequeued from out_i and used to update a location at Mem. This also elicits a message from Mem to each cache, via the respective in FIFOs.

- MemoryRead$_i(a)$. A cache line is sent from Mem to $cache_i$ by appending to in_i.

- CacheUpdate$_i(a, d)$. A new cache line is brought into $cache_i$ by dequeuing a message from in_i.

- CacheInvalidate$_i(a)$. Here $cache_i$ nondeterministically invalidates one of its addresses.

An operational description of Lazy Caching is given in Fig. 2.3 using guarded commands.

Lazy Caching has received considerable interest from the formal verification community. A special issue of the journal Distributed Computing, dedicated to proof techniques for this protocol, has been published [99]. Arons also considers Lazy Caching for her proof technique [11]. The challenge in verifying SC of Lazy Caching stems from the fact that the the reorderings involved to "witness" SC are relatively convoluted. In particular, Lazy Caching is not simple SC (defined formally in Def. 27), where writes to the same address cannot be reordered. Hence the following theorem supports the thesis that DSC has a high "reordering expressivity".

Event	Guard	Action
(R,i,j,d)	$cache_i[j]$ is valid $\wedge$ $cache_i[j] = d$ $\wedge$ no starred entries in in_i $\wedge$ out_i is empty	none
(W,i,j,d)	true	$out_i := append(out_i, (j,d))$
MemoryWrite$_i(a,d)$	$head(out_i) = (a,d)$	$Mem[a] := d$; $out_i := tail(out_i)$; $\forall k \neq i : in_k := append(in_k, (a,d))$; $in_i := append(in_i, (a,d,*))$
MemoryRead$_i(a)$	$cache_i[j]$ is invalid	$in_i := append(in_i, (a, Mem[a]))$
CacheUpdate$_i(a,d)$	$head(in_i) \in \{(a,d),(a,d,*)\}$	$in_i := tail(in_i)$; $cache_i(a) := d$; mark $cache_i(a)$ valid
CacheInvalidate$_i(a)$	$cache_i[j]$ is valid	mark $cache_i(a)$ invalid

Figure 2.3: Lazy Caching in guarded commands; this description is from Arons' paper [11].

Theorem 3. *Lazy Caching is DSC$_{r+n\ell+1}$, where r and ℓ are the respective capacities of the in and out FIFOs, and n is the number of processors.*

Proof. We prove this theorem on page 38, after we develop the *view window* machinery used in its proof. □

It is important to note that DSC$_{r+n\ell+1}$ contains many traces not admitted by Lazy Caching. In fact, even DSC$_2$ has traces with this property; in Example 3 of Sect. 2.5 we will see that the trace

$$(W,1,1,1)(R,1,1,1)(W,2,1,2)(R,2,1,1)$$

is DSC$_2$; this trace is given by Afek et al. as an example of a trace that Lazy Caching does not admit [6].

2.4 Complexity of Trace Problems

Given a set $A \subseteq \mathcal{M}^*$, the *trace problem for A* is the decision problem that asks, given trace τ, if $\tau \in A$. In this section, we look at the complexity of the trace problem for DSC and DSC$_k$. One might be interested in solving the trace problem for traces obtained by random or user-directed pre-silicon simulation or so-called post-silicon verification [70].

The trace problem for SC was explored by Gibbons and Korach [66], in which several quite restricted versions of the problem are proven to be NP-complete.[8] Here we show that the problem remains NP-complete when we consider DSC traces. In fact, we prove the stronger theorem that the trace problem for DSC_k is NP-complete for each $k \geq 2$. Our reduction is inspired by the reduction employed in the proof of Gibbons and Korach's Theorem 2.1 [66], however, in order that the resulting trace has DSC bound 2, several nontrivial modifications are necessary. First we show that the trace problem for DSC_1 can be solved in linear time; this result stems from the following important lemma.

Lemma 1. *DSC_1 are precisely the serial traces.*

Proof. Let τ be a trace of DSC_1. Then there exists a DSC-reordering π of τ such that shuffle degree of (τ, π, ℓ) is 1 for all values of ℓ. Clearly this holds if and only if π is the identity permutation. □

Theorem 4. *The trace problem for DSC_1 can be solved in linear time.*

Proof. From Lemma 1, the problem is equivalent to checking for seriality. A linear time algorithm simply scans the trace from left to right. Each address mentioned in the trace corresponds to an address used by the algorithm. Every time a write is encountered, the address is updated with the written value. Every time a read is encountered, the algorithm simply checks that the read value is the value currently stored at the relevant address. □

Our NP-completeness proof is quite involved; we first introduce a couple more definitions. Some of the proofs in this section (as well as those of future sections) involve reasoning how events from a trace τ' can be/must be DSC-reordered into a trace ${\tau'}^{\pi}$. We will desire certain substrings τ of τ' to be *atomic* in the sense that τ is always guaranteed to be a substring of ${\tau'}^{\pi}$. In other words, we would like to preclude the possibility that in ${\tau'}^{\pi}$, τ is "shuffled" with other events. This is achieved by a mechanism of Alur et al.'s [8], which requires the introduction of a fresh address lock.

[8] It should be noted that Gibbons and Korach [66] use a slightly different formalism. Rather than a single trace of memory events, their input is a set of traces in which each trace is the history of a single processor. The difference, then, is that they don't have any information about relative orderings of events at different processors. However, all their results apply to our "single trace" formalism; since the definition of SC doesn't call upon the relative ordering of events at different processors, we may simply transform one of our traces into a set of histories in the obvious way.

Definition 8 (atomicity construct). *Given a memory trace τ such that all events of τ are performed by the same processor p, define the* atomicity construct

$$\langle \tau \rangle \ = \ (W, p, \mathsf{lock}, p) \ \tau \ (R, p, \mathsf{lock}, p)$$

Intuitively, the atomicity construct has p write its name into lock before τ, and then read its name back after τ. Atomicity will only be guaranteed if all processors follow the same protocol. To avoid cluttering our presentation, we decree that all events e that are *not* explicitly nested in an atomicity construct will implicitly be replaced with $\langle e \rangle$, to ensure that the substrings that need to be atomic are such. Hence, from now on we tacitly assume that atomicity constructs yield atomicity. Also, we note that though Alur et al. [8] employ this construct for reasoning about sequential consistency, it adapts to DSC since all reads of lock inherit from previous writes. For brevity, sometimes we will omit processor names from the events inside an atomicity construct (since they are necessarily all the same), and subscript the construct itself with the processor name. For example $\langle (W, a, v)(R, a', v')(R, a'', v'') \rangle_p$ is shorthand for $\langle (W, p, a, v)(R, p, a', v')(R, p, a'', v'') \rangle$.

Our reduction involves the well-known NP-complete problem 3SAT, defined as follows. Let $X = \{x_1, \ldots, x_n\}$ be a set of boolean variables. A *literal* is a variable x_i or its negation $\overline{x_i}$, a *clause* is a disjunction of literals, and a *CNF formula* is a conjunction of clauses. Finally, a *3CNF formula* is simply a CNF formula in which all clauses have exactly 3 literals. The decision problem 3SAT asks, given a 3CNF formula φ, if there exists an assignment $\mathsf{assign} : X \to \mathbb{B}$ such that φ evaluates to T under assign. If such an assign exists, φ is said to be *satisfiable*, otherwise φ is *unsatisfiable*.

Given a 3CNF formula φ, we will construct a trace τ_φ such that φ is satisfiable if and only if τ_φ is DSC. Let $\{c_1, \ldots, c_m\}$ be the set of clauses of φ. Then τ_φ involves the following addresses.

- An address x_i for each variable $x_i \in X$. The values written to x_i are the booleans $\mathbb{B}$. We call these *variable addresses*.
- An address c_j for each clause c_j of φ. These addresses either store T (indicating the clause has been satisfied) or X (used to invalidate previous written Ts in the reordering). We call these *clause addresses*.

The processors of τ_φ are:

$$\begin{array}{l|lll} x_u^j & \left\langle (R, x_u, \mathsf{pol}_j(u))\ (W, c_j, \mathsf{T}) \right\rangle & & \\ x_v^j & & \left\langle (R, x_v, \mathsf{pol}_j(v))\ (W, c_j, \mathsf{T}) \right\rangle & \\ x_w^j & & & \left\langle (R, x_w, \mathsf{pol}_j(w))\ (W, c_j, \mathsf{T}) \right\rangle \end{array}$$

Figure 2.4: The trace $\mathsf{ec}(j)$.

- For each $x_i \in X$ and each $b \in \mathbb{B}$, a processor named x_i^b. This processor is responsible for writing value b to address x_i.
- A processor x_i^j for each i, j such that variable x_i appears in clause c_j.
- A processor named q, which is responsible for checking that all clauses are satisfied.

We build τ_φ by defining several of its constituent subtraces. For a variable $x_i \in X$ and $b \in \mathbb{B}$, define

$$\mathsf{eva}(i, b) \quad = \quad \left\langle \left(\prod_{j=1}^{m} (W, x_i^b, c_j, \mathsf{X}) \right) (W, x_i^b, x_i, b) \right\rangle$$

Intuitively, $\mathsf{eva}(i, b)$ assigns X to all clause addresses and writes b into address x_i, all atomically. eva stands for *encode variable assignment*.

For each clause c_j of φ, we define a trace $\mathsf{ec}(j)$ in Fig. 2.4. Here the variables of c_j are $x_u, x_v, x_w \in X$, and $\mathsf{pol}_j(u)$ is the polarity of the literal of x_u in c_j, i.e. $\mathsf{pol}_j(u) = \mathsf{F}$ if x_u is negated in c_j and $\mathsf{pol}_j(u) = \mathsf{T}$ if x_u is not negated in c_j. $\mathsf{pol}_j(v)$ and $\mathsf{pol}_j(w)$ are defined analogously for the variables x_v and x_w. The events of each row are all done by the processor named on the left, hence processor names are omitted from the events. Intuitively, $\mathsf{ec}(j)$ has atomicity constructs for three processors, and each will write T to address c_j if it occurs in the phase of a variable address that satisfies the clause c_j; ec stands for *encode clause*.

We may now define τ_φ. Note that the relative ordering of atomicity constructs across the three lines defining τ_φ will prove to be important, however many of the relative orderings within each line are arbitrary.

Definition 9 (**τ_φ**). *Given a 3CNF formula φ over variables $X = \{x_1, \ldots, x_n\}$ and with clauses $\{c_1, \ldots, c_m\}$, we define*

$$\begin{aligned} \tau_\varphi \quad = \quad & \textstyle\prod_{i=1}^{n} (\mathsf{eva}(x_i, \mathsf{F})\mathsf{eva}(x_i, \mathsf{T})) \\ & \textstyle\prod_{i=1}^{m} \mathsf{ec}(i) \\ & \langle \textstyle\prod_{i=1}^{m} (R, c_i, \mathsf{T}) \rangle_q \end{aligned}$$

Lemma 2. $|\tau_\varphi| = \mathcal{O}(|\varphi|^2)$.

Proof. Obvious. □

Our two key lemmas regarding the relationship between φ and τ_φ are now stated and proven.

Lemma 3. *If 3CNF formula φ is satisfiable, then τ_φ is DSC$_2$.*

Proof. Let $a : X \to \mathbb{B}$ be an assignment that satisfies φ. Given a clause c_j of φ, let $I_j, I'_j \subseteq \{1, \ldots, n\}$ be the indices of the variables with literals appearing in c_j that are satisfied by a (resp. unsatisfied by a), i.e. the set of (indices of) variables appearing in c_j is precisely $I_j \cup I'_j$. Note that since a satisfies φ it also satisfies each clause, thus we have $I_j \neq \emptyset$. Now define

$$\mathsf{ec}(j)^0 \;=\; \mathsf{ec}(j) \uparrow (*, \{x_i^j \mid i \in I_j\}, *, *)$$

and

$$\mathsf{ec}(j)^1 \;=\; \mathsf{ec}(j) \uparrow (*, \{x_i^j \mid i \in I'_j\}, *, *)$$

Note that it is possible that $I'_j = \emptyset$, hence $\mathsf{ec}(j)^1$ could be empty. Then let π be the DSC-reordering of τ_φ defined by

$$\tau_\varphi{}^\pi \;=\; \prod_{i=1}^{n} \mathsf{eva}(x_i, a(x_i)) \tag{2.2a}$$

$$\prod_{i=1}^{m} \mathsf{ec}(i)^0 \tag{2.2b}$$

$$\left\langle \prod_{i=1}^{m} (R, c_i, \mathsf{T}) \right\rangle_q \tag{2.2c}$$

$$\prod_{i=1}^{n} \mathsf{eva}(x_i, \overline{a(x_i)}) \tag{2.2d}$$

$$\prod_{i=1}^{m} \mathsf{ec}(i)^1 \tag{2.2e}$$

Here we place a bar over a member of $\mathbb{B}$ to obtain the other member. That π is a DSC-reordering follows from the facts that each clause address must be assigned T in (2.2b), hence the reads of (2.2c) are all satisfied, and all reads in (2.2e) inherit from events in (2.2d). Also, all reads of (2.2b) inherit from writes in (2.2a), since a satisfies every clause.

We now argue that (τ_φ, π) has DSC-bound 2. Consider colouring τ_φ such that a prefix is black and the remainder is white. In general, the induced striping on ${\tau_\varphi}^\pi$ will involve at most 2 black stripes, and if there are exactly 2, one of the black stripes will be at the start of ${\tau_\varphi}^\pi$. Let us call these black stripes b_1 and b_2, respectively. Note that in some cases these stripes can be empty.[9] b_1 starts at the beginning of ${\tau_\varphi}^\pi$, i.e. (2.2a), and b_2 starts at the beginning of (2.2d). It follows that the shuffle degree is always at most 2, and thus (τ_φ, π) has DSC-bound 2. □

Lemma 4. *If 3CNF formula φ is unsatisfiable, then τ_φ is not SC.*

Proof. Suppose τ_φ is SC. Consider any serial reordering π of τ_φ, and the placement of the atomicity construct $\sigma = \langle \prod_{i=1}^{m}(R, q, c_i, \mathsf{T})\rangle$ therein. Clearly, prior to σ, for each clause c_j of φ, there must exist a write of T to the address c_j. Now, consider the earliest (in ${\tau_\varphi}^\pi$) write w to a clause address c_j such that an event of σ inherits from w. Then ${\tau_\varphi}^\pi$ has the form

$$\beta_1 \; w \; \beta_2 \; \sigma \; \beta_3$$

for some traces β_1, β_2, and β_3. Now β_2 must not have any writes to variable addresses, since any such write occurs in an atomicity construct $\mathsf{eva}(i, b)$ which includes events that overwrite all clause addresses with X, and hence σ would not inherit from w. Further, the trace $w\beta_2$ contains a write of T to every clause address. Define $a : X \to \mathbb{B}$ such that for each $x_i \in X$, $a(x_i)$ is the value written in the rightmost (of the at least one and at most two) write to address x_i in β_1. It follows that all clauses are satisfied by a, and thus φ is satisfiable. □

We may now state and prove our desired results.

Theorem 5. *The following are NP-complete*

- *the trace problem for DSC$_k$, where $k \geq 2$.*
- *the trace problem for DSC*

[9] b_2 will be empty in two cases: 1) all of τ_φ is black, or 2) the black prefix is so short as to only involve events of $\mathsf{eva}(x_1, \mathsf{F})$, and also $a(x_1) = \mathsf{F}$. Furthermore, there is a case in which the beginning of ${\tau_\varphi}^\pi$ is white, i.e. b_1 is empty. This occurs when the black prefix is so short as to only involve events of $\mathsf{eva}(x_1, \mathsf{F})$, and $a(x_1) = \mathsf{T}$, which results in all black events from the beginning of τ_φ contributing to b_2.

Proof. First we note that any of these problems is in NP. For a trace τ, the witness is a DSC-reordering π, with the additional requirement that (τ, π) has DSC-bound k if we are checking DSC$_k$. It is straightforward to check, in polynomial time, if a purported witness π has the necessary properties.

We now show that the trace problem for DSC$_2$ is NP-hard, by reducing 3SAT to this problem. From Lemmas 3 and 4, given an instance φ of 3SAT, we may construct a trace τ_φ such that if φ is satisfiable, then τ_φ is DSC$_2$, otherwise τ_φ is not even SC. Furthermore, from Lemma 2, τ_φ has length polynomial in $|\varphi|$, and clearly it can be constructed in time proportional to its length.

Finally, because of the following containments

$$\mathrm{DSC}_2 \subseteq \mathrm{DSC}_3 \subseteq \mathrm{DSC}_4 \subseteq \cdots \subseteq \mathrm{DSC} \subseteq \mathrm{SC}$$

the same reduction works for DSC$_k$ where $k > 2$, as well as for unbounded DSC, hence these problems are also NP-hard. □

2.5 Complexity of Model Checking DSC$_k$

This section is devoted to proving that for any fixed k, the problem of checking if a given protocol $\mathcal{P}$ is DSC$_k$ is decidable. The main result is Theorem 8, which states that this problem is in EXPSPACE.

Let $\mathcal{P}$ be a protocol involving n processors, m addresses, and v data values, let k be a positive integer, and suppose we wish to decide if $\mathcal{P}$ is DSC$_k$. Let DSC$_k(n, m, v)$ be the set of all DSC traces with DSC-bound k over $\mathcal{M}(\mathbb{N}_n, \mathbb{N}_m, \mathbb{N}_v)$. We prove Theorem 8 by showing that there exists a protocol $\mathcal{G}_k(n, m, v)$ such that

- $\mathit{traces}(\mathcal{G}_k(n, m, v)) = \mathrm{DSC}_k(n, m, v)$, and
- the number of states of $\mathcal{G}_k(n, m, v)$ is $2^{\mathcal{O}(|\mathcal{P}| \log |\mathcal{P}|)}$.

Thus, we may decide our problem by checking the finite automata language containment $\mathit{traces}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{G}_k(n, m, v))$. Since deciding finite automata language containment is PSPACE-complete [122], and our containment instance is exponential in the size of our original query (i.e. the input of which is $\mathcal{P}$), the implied algorithm executes in exponential space. An important notion in defining the state space of $\mathit{traces}(\mathcal{G}_k(n, m, v))$ is that of the *view window*, which is expounded on in Sect. 2.5.1.

We now proceed with the details of the proof. For the time being, let us fix n, m, and v, so that $\mathcal{M}$ represents $\mathcal{M}(\mathbb{N}_n, \mathbb{N}_m, \mathbb{N}_v)$, and *trace* will mean *string over this* $\mathcal{M}$. As motivation, we first describe an infinite-state protocol, $\mathcal{G}_\infty$, whose trace set is the set of all DSC traces, and then consider how the state space of $\mathcal{G}_\infty$ could be made finite. Let $\mathcal{G}_\infty = (S_\infty, \mathcal{M}, \delta_\infty, \epsilon)$, where

- S_∞ is the (countably infinite) set of all serial traces over $\mathcal{M}$
- $\delta_\infty \subseteq S \times \mathcal{M} \times S$ includes exactly those triples $(\sigma, \alpha, \sigma')$ where σ' is a serial trace obtained by inserting α at some position j in σ, where $j \geq lpa(\sigma, proc(\alpha))$.
- ϵ is the empty trace

Then, it can be shown by induction on $|\tau|$ that

Claim 1. $(\epsilon, \tau, \sigma) \in \delta_\infty^*$ *if and only if* σ *is a DSC reordering of* τ.

Our claim implies that the trace set of the protocol $\mathcal{G}_\infty$ is exactly the set of DSC traces. Intuitively, to obtain a (restricted) finite state version of $\mathcal{G}_\infty$, it is necessary to avoid storing all of a serial trace σ in the state. For this purpose, in Sect. 2.5.1 we develop a data structure we call a *view window* that is a condensed version of a serial trace. View windows generalize Braun et al.'s concept of the *window* [22]. Also in Sect. 2.5.1 we define operations that morph view windows; these operations roughly correspond to dropping information, performing memory events, and advancing processors in logical time.

2.5.1 View Windows

A view window is an abstraction of a serial trace σ that can be used to determine, for a few positions j and any memory action α, whether the string σ' obtained from σ by inserting α at position j is serial. To this end, a view window contains an array of bookkeeping information called a *view* (of position j). For each address b, the view contains the following information:

- The value of the last write operation to b before position j in σ. This is used to determine the acceptable value of a read operation to address b, inserted at position j.
- A pair of tags. One tag indicates whether or not the predicate $IR(\sigma, j, b)$ holds; that is, whether position j is between a write to address b in σ and a read that inherits from the write.

This tag can be used to determine whether the protocol is free to insert a write operation to address b at position j. The tag has value F (**f**ree to write) or O (reads **o**nly). The other tag is needed to help update the F/O tag. It has value L (**l**ast) or N (**n**ot last).

In addition to the views, the view window contains additional information, called the *processor position function*, which is used to determine whether insertion of α respects processor order.

We now formally define view windows. Define the set of *tagged values* $T = (\mathbb{N}_v \cup \{\bot\}) \times \{L, N\} \times \{O, F\}$. Given a tagged value $z = (v, t_1, t_2)$, we extract the components via $val(z) = v$, $tag_1(z) = t_1$, and $tag_2(z) = t_2$. A *view* is a function $\nu \colon \mathbb{N}_m \to T$. Given a serial trace $\sigma \in \mathcal{M}^*$, a (non-contiguous) subsequence ps of $0, \ldots, |\sigma|$ such that ps includes $|\sigma|$ is called a *position sequence (for σ).*

Definition 10 (VW-set,VW). *Given serial trace σ and a position sequence ps for σ, we define the* VW-set $VW(\sigma, \mathsf{ps})$ *to be the set of all pairs* $(\mathsf{v}, \mathsf{lp})$ *that satisfy the following:* v *is a sequence of views with* $|\mathsf{v}| = |\mathsf{ps}|$ *defined as follows.*

1. $val(\mathsf{v}(i)(b)) = val(\sigma(lw(\sigma, \mathsf{ps}(i), b)))$
2. $tag_1(\mathsf{v}(i)(b)) = \begin{cases} L & \text{if } i = 1 \vee [i > 1 \wedge \mathsf{ps}(i-1) < lw(\sigma, \mathsf{ps}(i), b)] \\ N & \text{otherwise} \end{cases}$
3. $tag_2(\mathsf{v}(i)(b)) = \begin{cases} O & \text{if } IR(\sigma, \mathsf{ps}(i), b) \\ F & \text{otherwise} \end{cases}$

and lp*, called the* processor position function *is a function* $\mathbb{N}_n \to \mathbb{N}_{|\mathsf{v}|}$ *which must satisfy*

4. $\mathsf{ps}(\mathsf{lp}(p)) \geq lpa(\sigma, p)$ *for all* $p \in \mathbb{N}_n$.

A pair $w = (\mathsf{v}, \mathsf{lp})$ *is called a* view window (VW) *(of σ) if there exists* ps *such that* $(\mathsf{v}, \mathsf{lp}) \in VW(\sigma, \mathsf{ps})$. *The* size *of a VW* $w = (\mathsf{v}, \mathsf{lp})$ *is defined to be* $|w| = |\mathsf{v}|$.

Example 1. *An example of a serial trace $\sigma \in \mathcal{M}(\mathbb{N}_3, \mathbb{N}_2, \mathbb{N}_2)^*$, a position sequence ps for σ, and a VW $(\mathsf{v}, \mathsf{lp}) \in VW(\sigma, \mathsf{ps})$ are given in Fig. 2.5. Fig. 2.5(a) is the trace σ. The top row of Fig. 2.5(b) are the indices of ps, the second row is ps, the third row is v (in which each view ν is expressed as a column vector $[\nu(1)\ \ \nu(2)]^T$), and the fourth row specifies lp. All entries in Fig. 2.5(b) are aligned vertically with their corresponding position in σ.*

Recall that in the infinite-state protocol $\mathcal{G}_\infty$ (discussed on page 27), upon each transition, the state (a serial trace) is updated by insertion of a memory action. In a protocol that abstracts states as view windows, we need operations (state transitions) that update view windows upon insertion of a memory action. We define five such operations as functions that take VWs to VWs. The $delete$ function removes a view from a view window (in order to keep the view small). The $insert$ function inserts a view into a view window (corresponding to the insertion of a memory action α into the serial trace that the view window abstracts). The hop function makes it possible to advance $\mathsf{lp}(p)$ for any processor p (in order that an operation of processor p is inserted after the appropriate view). Finally, the $unfree$ and $bind$ operations update the view tags appropriately. Example applications of these functions are given in Example 2.

We now formally define the five operations. In all descriptions, $(\mathsf{v}', \mathsf{lp}')$ is the VW returned by the function, and p and b may be any processor and address, respectively.

- $delete((\mathsf{v}, \mathsf{lp}), h)$ requires $h \in \mathbb{N}_{|\mathsf{v}|-1}$. The new view v' is defined according to

$$\mathsf{v}'(j)(b) = \begin{cases} \mathsf{v}(j)(b) & \text{if } j < h \\ (val(\mathsf{v}(h+1)(b)), L, tag_2(\mathsf{v}(h+1)(b))) & \\ & \text{if } j = h \\ & \quad \wedge\ tag_1(\mathsf{v}(h)(b)) = L \\ & \quad \wedge\ tag_1(\mathsf{v}(h+1)(b)) = N \\ \mathsf{v}(j+1)(b) & \text{otherwise} \end{cases}$$

  Finally, for all $p \in \mathbb{N}_n$ we have $\mathsf{lp}'(p) = \mathsf{lp}(p)$ if $\mathsf{lp}(p) \leq h$, and $\mathsf{lp}'(p) = \mathsf{lp}(p) - 1$ otherwise.

- $hop((\mathsf{v}, \mathsf{lp}), p, h)$ requires $\mathsf{lp}(p) < h \leq |\mathsf{v}|$. hop leaves v unchanged, i.e. $\mathsf{v}' = \mathsf{v}$. lp' is everywhere equal to lp with the exception $\mathsf{lp}'(p) = h$.

- $insert((\mathsf{v}, \mathsf{lp}), p, \nu)$ is called against a view ν. Intuitively, v' is obtained from v by inserting ν at position $\mathsf{lp}(p)$. lp' is defined by $\mathsf{lp}'(q) = \mathsf{lp}(q)$ if $\mathsf{lp}(q) \leq \mathsf{lp}(p) \wedge p \neq q$; otherwise $\mathsf{lp}'(q) = \mathsf{lp}(p) + 1$.

- $unfree((\mathsf{v}, \mathsf{lp}), p, b)$: Let[10]

$$j = \max(\{i \mid i < \mathsf{lp}(p) \wedge tag_1(\mathsf{v}(i)(b)) = L\}).$$

[10] $unfree$ is only used when $\mathsf{lp}(p) > 1$ and Def. 10.2 ensures that $tag_1(\mathsf{v}(1)(b)) = L$, hence j is well-defined.

Then v' is everywhere equal to v, with the possible exceptions: for all $k \in \{j, \ldots, \mathsf{lp}(p) - 1\}$ we have $\mathsf{v}'(k)(b) = (val(\mathsf{v}(k)(b)), tag_1(\mathsf{v}(k)(b)), O)$. Finally, lp' is simply equal to lp.

- $bind((\mathsf{v}, \mathsf{lp}), p, b, v)$: Here we require $v \in \mathbb{N}_v$. This function leaves lp unchanged, and if $\mathsf{lp}(p) = |\mathsf{v}|$, then $\mathsf{v}' = \mathsf{v}$. Otherwise, let j be minimal such that $j > \mathsf{lp}(p)$ and $tag_1(\mathsf{v}(j)(b)) = L$; if no such j exists take $j = |\mathsf{v}| + 1$. Then v' is the same as v, with the exception that for all $k \in \{\mathsf{lp}(p) + 1, \ldots, j - 1\}$ we have $\mathsf{v}'(k)(b) = (v, N, F)$.

We define the partial order $\leq_{vw}$ on VWs such that $x \leq_{vw} y$ iff y can be obtained from x by 0 or more $delete$ and hop operations.

Similarly to the protocol $\mathcal{G}_\infty$, wherein the protocol "evolves" on memory actions from serial trace to serial trace, we can also define what it means to evolve on a memory action from VW to VW.

Definition 11. *Given VWs $w = (\mathsf{v}, \mathsf{lp})$ and w' and $\alpha \in \mathcal{M}$, we say that w* can directly α-evolve to *w' if the following conditions are satisfied.*

1. *If $\alpha = (R, p, b, v)$, then we require $v = val(\mathsf{v}(\mathsf{lp}(p))(b)) \neq \bot$, and the following equation must hold:*

$$w' = unfree(insert(w, p, \nu), p, b)$$

   where ν is given by

   $\nu(a) = (val(\mathsf{v}(\mathsf{lp}(p))(a)), N, tag_2(\mathsf{v}(\mathsf{lp}(p))(a)))$ *for each $a \in \mathbb{N}_m$.*

2. *If $\alpha = (W, p, b, v)$, then we must have*

   (a) $tag_2(\mathsf{v}(\mathsf{lp}(p))(b)) = F$

   (b) $w' = bind(insert(w, p, \nu), p, b, v)$, *where ν is given by*

$$\nu(a) = \begin{cases} (v, L, F) & \text{if } a = b \\ (val(\mathsf{v}(\mathsf{lp}(p))(a)), N, tag_2(\mathsf{v}(\mathsf{lp}(p))(a))) & \\ & \text{otherwise} \end{cases}$$

If there exists w' such that w directly α-evolves to w', we say that w is directly α-enabled. *If there exist $w_1 = (\mathsf{v}_1, \mathsf{lp}_1)$ and $w_2 = (\mathsf{v}_2, \mathsf{lp}_2)$ such that $w \leq_{vw} w_1$ and $w_2 \leq_{vw} w'$, and w_1 directly*

(a)	σ		$(W,2,1,2)$		$(W,1,1,1)$	$(W,1,2,1)$	$(R,1,1,1)$		$(R,1,2,1)$				$(W,2,2,2)$	$(R,2,1,1)$		$(R,1,2,2)$	
	j	1		2				3		4					5		6
	$\mathsf{ps}(j)$	0		1				4		5					7		8
(b)	$\mathsf{v}(j)$	$\begin{bmatrix}\bot, LF\\ \bot, LF\end{bmatrix}$		$\begin{bmatrix}2, LF\\ \bot\end{bmatrix}$				$\begin{bmatrix}1, LO\\ 1, LO\end{bmatrix}$		$\begin{bmatrix}1, NO\\ 1, NF\end{bmatrix}$					$\begin{bmatrix}1, NF\\ 2, LO\end{bmatrix}$		$\begin{bmatrix}1, NF\\ 2, NF\end{bmatrix}$
	$\mathsf{lp}^{-1}(j)$	$\emptyset$		$\{3\}$				$\emptyset$		$\emptyset$					$\{2\}$		$\{1\}$
	j	1		2				3		4							5
	$\mathsf{ps}_1(j)$	0		1				4		5							8
(c)	$\mathsf{v}_1(j)$	$\begin{bmatrix}\bot, LF\\ \bot, LF\end{bmatrix}$		$\begin{bmatrix}2, LF\\ \bot\end{bmatrix}$				$\begin{bmatrix}1, LO\\ 1, LO\end{bmatrix}$		$\begin{bmatrix}1, NO\\ 1, NF\end{bmatrix}$							$\begin{bmatrix}1, NF\\ 2, LF\end{bmatrix}$
	$\mathsf{lp}_1^{-1}(j)$	$\emptyset$		$\emptyset$				$\emptyset$		$\{3\}$							$\{1,2\}$
(d)	σ_2		$(W,2,1,2)$		$(W,1,1,1)$	$(W,1,2,1)$	$(R,1,1,1)$		$(R,1,2,1)$		$(W,3,2,4)$		$(W,2,2,2)$	$(R,2,1,1)$		$(R,1,2,2)$	
	j	1		2				3		4		5					6
	$\mathsf{ps}_2(j)$	0		1				4		5		6					9
(e)	$\mathsf{v}_2(j)$	$\begin{bmatrix}\bot, LF\\ \bot, LF\end{bmatrix}$		$\begin{bmatrix}2, LF\\ \bot\end{bmatrix}$				$\begin{bmatrix}1, LO\\ 1, LO\end{bmatrix}$		$\begin{bmatrix}1, NO\\ 1, NF\end{bmatrix}$		$\begin{bmatrix}1, NO\\ 4, LF\end{bmatrix}$					$\begin{bmatrix}1, NF\\ 2, LF\end{bmatrix}$
	$\mathsf{lp}_2^{-1}(j)$	$\emptyset$		$\emptyset$				$\emptyset$		$\emptyset$		$\{3\}$					$\{1,2\}$

Figure 2.5: VW and VW operations referred to in Examples 1 and 2

α-evolves to w_2, we say w α-evolves to *w', denoted $w \rightsquigarrow_\alpha w'$. If there exists w' such that $w \rightsquigarrow_\alpha w'$ we say that w is α*-enabled.

Example 2. *Continuing with Example 1, let w be the VW of Fig. 2.5(b). Then Fig. 2.5(c) gives the VW $w_1 = (\mathsf{v}_1, \mathsf{lp}_1) = hop(delete(w, 5), 3, 4)$, along with position sequence ps_1 such that $w_1 \in VW(\sigma, \mathsf{ps}_1)$. Fig. 2.5(d) is a serial string σ_2 obtained by performing an insertion of $(W, 3, 2, 4)$ against σ. Finally, Fig. 2.5(e) presents a position sequence ps_2 for σ_2 and a VW $w_2 = (\mathsf{v}_2, \mathsf{lp}_2) \in VW(\sigma_2, \mathsf{ps}_2)$, where w_1 directly $(W, 3, 2, 4)$-evolves to w_2. Some other points of note here are: $w \rightsquigarrow_{(W,3,2,4)} w_2$ and $w \leq_{vw} w_1$. Also, the set of all α such that w_1 is directly α-enabled is*

$$(W, 1, *, *) \cup (W, 2, *, *) \cup (W, 3, 2, *) \cup$$
$$\{(R, 1, 1, 1), (R, 1, 2, 2), (R, 2, 1, 1), (R, 2, 2, 2), (R, 3, 1, 1), (R, 3, 2, 1)\}$$

This completes our description of operations on view windows and the associated α-evolves relation. Before proving properties of this relation in the rest of this section, we provide some informal intuition. First we note that we can generalize the notion of α-evolves to sequences τ of memory actions, where $w \rightsquigarrow_\tau w'$ if and only if there is a sequence $w_0, w_1, \ldots w_{|\tau|}$ of view windows such that $w = w_0$, $w' = w_{|\tau|}$, and $w_{i-1} \rightsquigarrow_{\tau(i)} w_i$ for $1 \leq i \leq |\tau|$. Let w_ϵ be the view window of the empty trace, that is, $w_\epsilon = (\mathsf{v}_0, \mathsf{lp}_0)$ where v_0 is a singleton view sequence with $\mathsf{v}_0(b) = (\perp, L, F)$ for each address b and $\mathsf{lp}_0(p) = 1$ for all $p \in \mathbb{N}_n$. It turns out that

Claim 2. *For all traces τ, $w_\epsilon \rightsquigarrow_\tau w$ if and only if τ is DSC and w is a view window of τ^π for some DSC reordering π of τ.*

Claim 2 is analogous to Claim 1 for serial traces given on page 27, except the former pertains to view windows. Roughly, Claim 2 can be proved by induction on $|\tau|$, using the following theorem.

Theorem 6. *Let τ have DSC-reordering π, w be a VW of τ^π, and $\alpha \in \mathcal{M}$ be such that $w \rightsquigarrow_\alpha w'$. Then there exists a DSC-reordering π' of $\tau' = \tau\alpha$ such that $\tau^{\pi'} = \tau^\pi$, and w' is a VW of ${\tau'}^{\pi'}$.*

Proof. From Def. 11, there exist VWs w_1 and w_2 such that $w \leq_{vw} w_1$ and $w_2 \leq_{vw} w'$, and w_1 directly α-evolves to w_2. From Lemma 5 we have that w_1 is a VW of τ^π. Choose ps to be a position sequence such that $w_1 = (\mathsf{v}_1, \mathsf{lp}_1) \in VW(\tau^\pi, \mathsf{ps})$, and define $(o, p, b, v) = \alpha$. Let π' be the permutation of τ' such that ${\tau'}^{\pi'}$ is the string obtained by inserting α into τ^π at position $\mathsf{lp}_1(p)$. From this definition of π' we have $\tau^{\pi'} = \tau^\pi$.

We must show that π' is a DSC-reordering of τ'. π' adheres to the processor order of τ', since π adheres to the processor order of τ, and by the Def. 10 we have $\mathsf{ps}(\mathsf{lp}(p)) \geq lpa(\tau^\pi, p)$, and Lemma 6 guarantees that ${\tau'}^{\pi'}$ is serial. Finally, from Lemma 7 we have that w_2 is a VW of ${\tau'}^{\pi'}$, and hence by Lemma 5 we have w' is a VW of ${\tau'}^{\pi'}$. □

Lemma 5. *If w is a VW of σ, and $w \leq_{vw} w'$ then w' is a VW of σ.*

Proof. We show that if w is a VW of σ, then applying a single hop or $delete$ operation gives another VW of σ; the lemma follows by induction. Let ps be such that $w \in VW(\sigma, \mathsf{ps})$.

Case: $w' = (\mathsf{v}', \mathsf{lp}') = hop(w, p, h)$. Then we have $\mathsf{lp}(p) < \mathsf{lp}'(p) = h \leq |\mathsf{v}'|$, this being the sole discrepancy between w and w'. Let $x = \min(\{i \mid lpa(\sigma, p) \leq \mathsf{ps}(i)\} \cup \{0\})$. Since $w \in VW(\sigma, \mathsf{ps})$, we have from Def. 10 that $\mathsf{lp}(p) \geq x$, and thus also $\mathsf{lp}'(p) \geq x$. Thus $w' \in VW(\sigma, \mathsf{ps})$.

Case: $w' = (\mathsf{v}', \mathsf{lp}') = delete(w, h)$. Let ps' be ps with the hth entry removed. We claim that $w' \in VW(\sigma, \mathsf{ps}')$, and show that v' and lp' are compliant with the conditions of Def. 10. Note that since $delete$ requires $h < |\mathsf{v}|$, we have that ps' has $|\sigma|$ as the final element, and hence ps' is a legal position sequence for σ. Let $\hat{\mathsf{v}}$ be v with $\mathsf{v}(h)$ removed, and let ps' be ps with the hth entry removed. Clearly, for all $i \in \mathbb{N}_{|\mathsf{ps}'|}$ we have $val(\hat{\mathsf{v}}(i)(b)) = \sigma(lw(\sigma, \mathsf{ps}'(i), b))$. Further, $tag_2(\hat{\mathsf{v}}(i)(b)) = O$ iff $IR(\sigma, \mathsf{ps}'(i), b)$. Thus v' and ps' are compliant with Def. 10(1) and Def. 10(3), since v' only differs from $\hat{\mathsf{v}}$ in tag_1 components. In general $\hat{\mathsf{v}}$ does not satisfy Def. 10(2). This is because the boolean expression in Def. 10(2) depends on other entries in the position sequence, which was altered. The only case where $\hat{\mathsf{v}}$ can potentially be in violation is at position h, which can be seen as follows. We have $\hat{\mathsf{v}}(h) = \mathsf{v}(h+1)$ and $\mathsf{ps}'(h) = \mathsf{ps}(h+1)$. It is possible that for some address b, we have that

$$\mathsf{ps}(h-1) < lw(\sigma, \mathsf{ps}(h), b) = lw(\sigma, \mathsf{ps}(h+1), b)$$

(taking $\mathsf{ps}(0) = 0$ if necessary) which implies both $tag_1(\hat{\mathsf{v}}(h)(b)) = N$ and $\mathsf{ps}'(h-1) < lw(\sigma, \mathsf{ps}'(h), b)$. In this case we find a noncompliance. However, $delete$ returns v', which differs from $\hat{\mathsf{v}}$ by correcting precisely such scenarios. Therefore v' is compliant. Since lp' is obtained from lp by simply decrementing values where appropriate to reflect the deleted view, it follows that lp' adheres to Def. 10(4). Again we conclude that $w' \in VW(\sigma, \mathsf{ps})$. □

Lemma 6. *Suppose $w = (\mathsf{v}, \mathsf{lp}) \in VW(\sigma, \mathsf{ps})$ and w is directly α-enabled. Then the string σ' obtained by inserting α at position $\mathsf{ps}(\mathsf{lp}(proc(\alpha)))$ in σ is serial.*

Proof. Since σ is serial, we must argue that the insertion of $\alpha = (o, p, b, v)$ maintains seriality. We case-split on $op(\alpha)$. Suppose α is a read. From Def. 11 we have $val(\mathsf{v}(\mathsf{lp}(p))(b)) = v \neq \bot$, and from Def. 10 we have that $val(\mathsf{v}(\mathsf{lp}(p))(b)) = val(lw(\sigma, \mathsf{ps}(\mathsf{lp}(p)), b))$. Thus σ' maintains seriality in this case. Now assume α is a write. From Def. 11 we have $tag_2(\mathsf{v}(\mathsf{lp}(p))(b)) = F$, hence from Def. 10 we have $\neg IR(\sigma, \mathsf{ps}(\mathsf{lp}(p)), b)$. Thus the insertion of α will not interfere with any inheritance from $lw(\sigma, \mathsf{ps}(\mathsf{lp}(p)), b)$, or any other write to address b. Therefore σ' is serial in both cases. □

Lemma 7. *Suppose $w = (\mathsf{v}, \mathsf{lp}) \in VW(\sigma, \mathsf{ps})$ and w directly α-evolves to w'. Letting $p = proc(\alpha)$, let σ' be obtained by inserting α at position $\mathsf{ps}(\mathsf{lp}(p))$, and let ps' be the position sequence of σ' obtained by inserting $\mathsf{ps}(\mathsf{lp}(p)){+}1$ into ps at $\mathsf{lp}(p)$, and then incrementing all entries right of $\mathsf{lp}(p)$.*[11] *Then $w' \in VW(\sigma', \mathsf{ps}')$.*

Proof. Let $(o, p, b, v) = \alpha$. We note that $VW(\sigma', \mathsf{ps}')$ is well-defined, since from Lemma 6 we have σ' serial. We case split on o. For convenience in this proof, we introduce the function $c : \mathbb{N}_{|\mathsf{ps}'|} \rightarrow \mathbb{N}_{|\mathsf{ps}|}$ given by $c(i) = i$ when $i \leq \mathsf{lp}(p)$ or $c(i) = i - 1$ otherwise. Also, ν is the inserted view, as in Def. 11.

Case $o = R$: Clearly Def. 10(1) is satisfied w.r.t v', since for all addresses a we have $val(\nu(a)) = val(\mathsf{v}(\mathsf{lp}(p))(a))$. This follows from the fact that $lw(\sigma', \mathsf{ps}'(\mathsf{lp}'(p)), a) = lw(\sigma, \mathsf{ps}(\mathsf{lp}(p)), a)$, which also implies that Def. 10(2) is satisfied, since for all addresses a we have $tag_1(\nu(a)) = N$ and $\mathsf{lp}'(p) > 1$. Now for all addresses $a \neq b$ and indexes i of ps', we have $IR(\sigma', \mathsf{ps}'(i), a) \Leftrightarrow IR(\sigma, \mathsf{ps}(c(i)), a)$. Thus, the fact that for all such a, we have $tag_2(\mathsf{v}'(i)(a)) = tag_2(\mathsf{v}(c(i))(a))$ is compliant with Def. 10(3). Also, if $\ell = lr(\sigma, lw(\sigma, \mathsf{lp}(p), b)) > \mathsf{ps}(\mathsf{lp}(p))$, then we have both $IR(\sigma', \mathsf{ps}'(i), b) \Leftrightarrow IR(\sigma, \mathsf{ps}(c(i)), b)$ and $tag_2(\mathsf{v}'(i)(b)) = tag_2(\mathsf{v}(c(i))(b))$ for all indexes i of ps'. Then Def. 10(3) is satisfied. However, if $\ell \leq \mathsf{ps}(\mathsf{lp}(p))$ then for each i in the nonempty set $I = \{i \mid i < \mathsf{lp}'(p) \wedge \ell \leq \mathsf{ps}'(i)\}$ we have $IR(\sigma', \mathsf{ps}'(i), b) \wedge \neg IR(\sigma, \mathsf{ps}(c(i)), b)$. In this case *unfree* has the effect that for all $i \in I$, $tag_2(\mathsf{v}'(i)(b)) = O$, which hence preserves compliance with Def. 10(3). Finally, we have Def. 10(4) satisfied w.r.t ps' and lp' since $\mathsf{ps}'(\mathsf{lp}'(p)) = \mathsf{ps}(\mathsf{lp}(p)){+}1 = lpa(\sigma', p)$. A similarly simple argument handles the other processors.

Case $o = W$: Similar to the previous case, Def. 10(1) is satisfied w.r.t v'. The slight complication here is that for all i in $I = \{\mathsf{lp}'(p), \ldots, j_{min}\}$ where

[11] *I.e.* $\mathsf{ps}' = \mathsf{ps}(1), \ldots, \mathsf{ps}(\mathsf{lp}(p)), \mathsf{ps}(\mathsf{lp}(p)) + 1, \ldots, \mathsf{ps}(|\mathsf{ps}|) + 1$

$j_{min} = \min(\{j \mid j > \mathsf{lp}'(p) \wedge tag_1(\mathsf{v}'(j)(b)) = L\} \cup \{|\mathsf{v}'| + 1\}) - 1$ we have $lw(\sigma', \mathsf{ps}'(i), b) = \mathsf{ps}'(\mathsf{lp}'(p)) \neq lw(\sigma, \mathsf{ps}(c(i)), b)$. However, Def. 11(2) correctly sets $val(\nu(b)) = val(\sigma'(\mathsf{ps}'(\mathsf{lp}'(p)))) = v$, and $bind$ sets $val(\mathsf{v}'(i)(b)) = v$ for all $i \in I$. Now for all addresses $a \neq b$, we have $lw(\sigma', \mathsf{ps}'(\mathsf{lp}'(p)), a) = lw(\sigma, \mathsf{ps}(\mathsf{lp}(p)), a)$, and since we have $tag_1(\nu(a)) = N$ and $\mathsf{lp}'(p) > 1$, Def. 10(2) is satisfies w.r.t all such a. For address b, we have $lw(\sigma', \mathsf{ps}'(\mathsf{lp}'(p)), b) = \mathsf{ps}'(\mathsf{lp}'(p)) = \mathsf{ps}'(\mathsf{lp}'(p) - 1) + 1$, and thus, $\mathsf{ps}'(\mathsf{lp}'(p) - 1) < lw(\sigma', \mathsf{ps}'(\mathsf{lp}'(p)), b)$. Thus the assignment $tag_1(val(\nu(b))) = L$ is consistent with Def. 10(2). Similar to the previous case, we have Def. 10(3) satisfied. Def. 11(2) assigns $tag_2(\mathsf{v}'(i)(a)) = tag_2(\mathsf{v}(c(i))(a))$ for all addresses a and indexes i of ps', i.e. the tag_2 components are preserved. The correctness of this preservation follows from the fact that for all such a and i we have $IR(\sigma', \mathsf{ps}'(i), a) = IR(\sigma, \mathsf{ps}(c(i)), a)$. The argument that Def. 10(4) is satisfied is the same as in the previous case. For both cases, we conclude that $w' \in VW(\sigma', \mathsf{ps}')$. $\square$

2.5.2 VW-Boundedness

In this section we define the concept of the VW-bound of a DSC trace, and show that this notion corresponds to the DSC-bound.

Definition 12 (VW-bound). *Let τ be a trace such that there exists VWs $w_0, w_1, \ldots, w_{|\tau|}$ and $k \geq 1$ such that*

1. $w_0 = w_\epsilon$ *(the view window of the empty trace),*
2. *for all $1 \leq i \leq |\tau|$ we have $w_{i-1} \rightsquigarrow_{\tau(i)} w_i$, and*
3. *for all $1 \leq i \leq |\tau|$ we have $|w_i| \leq k$,*

Then we say that τ has VW-bound k.

Theorem 7. *A trace τ has VW-bound k if and only if τ is DSC$_k$.*

Proof. ($\Leftarrow$) Let π be a DSC-reordering of τ such that (τ, π) has DSC-bound k. For each $0 \leq \ell \leq |\tau|$, we define a position sequence ps_ℓ for $\tau[\ell]^\pi$. Consider the striping induced on τ^π by colouring the events of $\tau[\ell]$ black and the remaining suffix white. The colouring of τ^π defines ps_ℓ as follows. Note that $\tau[\ell]^\pi$ can be obtained from τ^π by removing all white events; we include index i in ps_ℓ

v_0	$(\bot, LF)$								
lp_0^{-1}	$\{1,2\}$								
$\tau[1]^\pi$		$(W,1,1,1)$							
v_1	$(\bot, LF)$		$(1, LF)$						
lp_1^{-1}	$\{2\}$		$\{1\}$						
$\tau[2]^\pi$		$(W,1,1,1)$		$(R,1,1,1)$					
v_2	$(\bot, LF)$				$(1, LF)$				
lp_2^{-1}	$\{2\}$				$\{1\}$				
$\tau[3]^\pi$		$(W,2,1,2)$		$(W,1,1,1)$		$(R,1,1,1)$			
v_3			$(2, LF)$				$(1, LF)$		
lp_3^{-1}			$\{2\}$				$\{1\}$		
$\tau[4]^\pi$		$(W,2,1,2)$		$(W,1,1,1)$		$(R,1,1,1)$		$(R,2,1,1)$	
v_4							$(1, LO)$		$(1, NF)$
lp_4^{-1}							$\{1\}$		$\{2\}$

Figure 2.6: The DSC_2 trace used in Example 3

whenever a nonempty segment of white events was removed directly following the ith event of $\tau[\ell]^\pi$. Also, we include 0 at the beginning of in ps_ℓ if the first event of τ^π is white, or if $\ell = 0$. Now let w_ℓ be the unique view window of $VW(\tau[\ell]^\pi, \mathsf{ps}_\ell)$ in which lp is such that for each processor p, $\mathsf{lp}(p)$ is minimal. Intuitively, this lp places each processor as early as possible. It can be shown that $w_{\ell-1} \rightsquigarrow_{\tau(\ell)} w_\ell$ for each $1 \leq \ell \leq |\tau|$. Also, our definitions imply that $w_0 = w_\epsilon$, and for each $1 \leq \ell \leq |\tau|$ we have $|w_\ell| \leq k$. Thus τ has VW-bound k.

($\Rightarrow$) Suppose τ has VW-bound k. Then there exist VWs $w_0, \ldots, w_{|\tau|}$ such that the three conditions of Def. 12 hold. Furthermore, by Theorem 6, there exists some DSC-reordering π of τ such that for each $0 \leq \ell \leq |\tau|$ we have that w_ℓ is a VW of $\tau[\ell]^\pi$. Now for any prefix length ℓ, factor $\tau = \tau[\ell]\tau'$. Then $w_\ell \rightsquigarrow_{\tau'} w_{|\tau|}$, and there exists some position sequence ps of $\tau[\ell]^\pi$ such that $w_\ell \in VW(\tau[\ell]^\pi, \mathsf{ps})$. Then clearly τ^π can be obtained from $\tau[\ell]^\pi$ by inserting traces only at the positions listed in ps. If we colour events of τ' white and those of $\tau[\ell]$ black, this implies that the shuffle degree of (τ, π, ℓ) is at most $|\mathsf{ps}|$, which is at most k. Since ℓ is arbitrary, we have shown that (τ, π) has DSC-bound k. $\square$

Example 3. *Here we consider the trace* $\tau = (W,1,1,1)(R,1,1,1)(W,2,1,2)(R,2,1,1)$*, given by Afek et al. [6] as an example of a SC sequence that is* not *a trace of Lazy Caching. This example shows that* τ *is DSC with VW-bound 2 (and, by Theorem 7 is thus* DSC_2*). A DSC-reordering of* τ *is* π*, where* $\tau^\pi = (W,2,1,2)(W,1,1,1)(R,1,1,1)(R,2,1,1)$*. Fig. 2.6 gives VWs* $w_0, \ldots, w_4$ *of*

$\tau[0]^\pi, \ldots, \tau[4]^\pi$ *respectively that satisfy the conditions of Def. 12 for* $k = 2$*.*

We now show how our VW theory along with Theorem 7 can be employed to decide DSC$_k$. Let us unfix n, m, and v, and employ the term (n, m, v)-*VW* to refer to a VW for n processors, m addresses, and v data values.

Definition 13 **($\mathcal{G}_k(n, m, v)$).** *For any* $n, m, v, k \geq 1$*, define the finite automaton* $\mathcal{G}_k(n, m, v) = (S, \mathcal{M}(\mathbb{N}_n, \mathbb{N}_m, \mathbb{N}_v), \delta, w_\epsilon)$ *such that*

- S *is the set of all* (n, m, v)*-VWs* w *such that* $|w| \leq k$
- $\delta = \{(w, \alpha, w') \mid w \rightsquigarrow_\alpha w'\}$
- w_ϵ *is the empty* (n, m, v)*-VW*

Lemma 8. *For any integers* $n, m, v, k \geq 1$*,* $\mathcal{L}(\mathcal{G}_k(n, m, v)) = DSC_k(n, m, v)$*.*

Proof. Follows from Theorem 7. □

Lemma 9. *For any* $n, m, v, k \geq 1$*,* $\mathcal{G}_k(n, m, v)$ *has no more than* $k^n(4v)^{m(k+1)}$ *states.*

Proof. There are $(4v)^m$ possible views, hence at most $(4v)^{m(k+1)}$ possible view sequences. Each may have at most k^n processor position functions, thus the states S of $\mathcal{G}_k(n, m, v)$ is such that $|S| \leq k^n(4v)^{m(k+1)}$. □

We are now equipped to prove our desired theorem about deciding DSC$_k$.

Theorem 8. *For each* $k \geq 1$*, the problem of checking if a given protocol* $\mathcal{P}$ *is DSC$_k$ is in EXPSPACE.*[12]

Proof. Give a protocol $\mathcal{P}$ involving n processors, m addresses, and v data values, we construct the protocol $\mathcal{G}_k(n, m, v)$, which, by Lemma 9 has at most $k^n(4v)^{m(k+1)}$ states. Clearly $n, m, v \leq |\mathcal{P}|$, thus the number of states of $\mathcal{G}_k(n, m, v)$ is bounded by

$$k^{|\mathcal{P}|}(4|\mathcal{P}|)^{|\mathcal{P}|(k+1)} = |\mathcal{P}|^{\mathcal{O}(|\mathcal{P}|)} = 2^{\mathcal{O}(|\mathcal{P}| \log |\mathcal{P}|)} \tag{2.3}$$

[12]Here we adopt the definition of Aaronson [1], which defines EXPSPACE to be the class of problems solvable on a deterministic Turing machine using space $\mathcal{O}(2^{p(n)})$ for some polynomial $p(n)$. Some authors (e.g. Du and Ko [47]) use EXPSPACE to refer to the smaller class obtained by restricting $p(n)$ to being a linear function.

Now, by Lemma 8, $\mathcal{P}$ is DSC$_k$ if and only if

$$traces(\mathcal{P}) \subseteq traces(\mathcal{G}_k(n, m, v)) \tag{2.4}$$

(2.4) requires checking language containment of (nondeterministic) finite automata, which can be done in space polynomial in the size of the involved automata. Hence the problem can be decided in space polynomial in (2.3), which implies that the space requirements are exponential in $\mathcal{O}(|\mathcal{P}| \log |\mathcal{P}|)$. □

We conclude this section by proving Theorem 3 (originally stated on page 20), the proof of which was postponed until after the development of the view window theory.

Theorem 3. *Lazy Caching is DSC$_{r+n\ell+1}$, where r and ℓ are the respective capacities of the in and out FIFOs, and n is the number of processors.*

Proof. (Sketch) That Lazy Caching is DSC follows from Theorem 2 (page 18), along with the facts that Lazy Caching is data independent and SC. The remainder sketches the proof that Lazy Caching is in DSC$_{r+n\ell+1}$. For any state of Lazy Caching, we associate a finite set of possible VWs. We define this set by placing restriction on an arbitrary member $w = (\mathsf{v}, \mathsf{lp})$. Given a trace τ of Lazing Caching, these sets can be used to construct a sequence of VWs satisfying the conditions to witness that τ has VW-bound $r + n\ell + 1$ (see Def. 12 on page 35). Thus, by Theorem 7 (page 35), τ is DSC$_{r+n\ell+1}$.

For convenience, we shift the index set of the sequence v (and hence the range of the function lp) leftwards by $r+1$, i.e. $\mathsf{v} = \mathsf{v}(-r) \dots \mathsf{v}(0) \dots \mathsf{v}(n\ell)$. The val components of $\mathsf{v}(0)$ always represents the contents of Mem. When $\mathsf{lp}(i) \leq 0$, this corresponds to $|in_i| = -\mathsf{lp}(i)$ and $|out_i| = 0$, where $|q|$ gives the number of messages in the FIFO q. The $n\ell$ possible views to the right of $\mathsf{v}(0)$ are to accommodate the views corresponding to the $n\ell$ potential out queue entries that may be buffered in the system at any given time.

Prior to performing an event (W, i, a, v), the operation $hop(w, i, h)$ must be performed for some $h \geq 0$.[13] The actual value of h is nondeterministic; the relative ordering of the views to the right of

[13] This may seem counterintuitive, since Lazy Caching allows write events to be performed by processor i even if in_i is nonempty. However, after performing (W, i, a, v), processor i cannot perform any read event until in_i is flushed of all entries present at the time the write took place.

$\mathsf{v}(0)$ reflect a "prediction" made regarding the order that the associated W events are seen by Mem via MemoryRead actions.

Other Lazy Caching actions result in the following updates to w. ReadRequest, WriteRequest, and CacheInvalidate produce no change. MemoryWrite causes no effective change, though under our shifted view naming convention, all names would be decremented. MemoryRead inserts a new view with the same val components as $\mathsf{v}(0)$ immediately following $\mathsf{v}(0)$. CacheUpdate$_i$ simply increments $\mathsf{lp}(i)$ if $\mathsf{lp}(i) < 0$, otherwise CacheUpdate$_i$ effects no change.

To see that $r + n\ell + 1$ is an upper bound on $|w|$, we note that the leftmost view in v can always be deleted (without impeding any possible future events) whenever we have $\mathsf{lp}(i) > -r$ for all processors i. □

2.6 Complexity of Model Checking DSC

In Sect. 2.5 we saw that for any finite k, it is decidable to check if a protocol is DSC$_k$. We have been unsuccessful in our attempts to determine the decidability of the analogous problem for (unbounded) DSC. Decidability would follow from the following conjecture, a variation of which was originally made in a paper of the author et al.'s [18].

Conjecture 1. *Any DSC protocol $\mathcal{P}$ is DSC$_k$ for some k, and further k is computable given a description of $\mathcal{P}$.*[14]

In this section we show that this problem is not *too* easy; specifically, it is PSPACE-Hard. The proof uses a reduction from the problem 3-QUANTIFIED BOOLEAN FORMULA [47], which is defined as follows.

3-QUANTIFIED BOOLEAN FORMULA can be viewed as a generalization of the problem 3SAT, defined in Sect. 2.4 on page 22. Here it is convenient to have our boolean variables belong to $X \cup Y$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$ are disjoint. Hence, by *variable* we now mean any member of $X \cup Y$. We will use x_i (resp. y_i) to refer to a member of X (resp. Y), use v to refer to an arbitrary member of $X \cup Y$, and use v_i to refer to a member of $\{x_i, y_i\}$. At times we commit a minor abuse by identifying 0 with F and 1 with T, e.g. the statement of Lemma 15 is guilty of this.

[14] Here "computable" is probably far too general of a term; it is likely that k is always bounded by the number of states and in practice this bound is very loose.

The notions of *literal*, *clause*, *CNF formula*, and *3CNF formula*, are the same as in Sect. 2.4. A *3QBF formula* is a quantified boolean formula of the form:

$$\forall x_n \exists y_n \cdots \forall x_1 \exists y_1 : f \tag{2.5}$$

where f is a 3CNF formula. For the remainder, we assume f has the m clauses $C_1, \ldots, C_m$. Each 3QBF formula evaluates to either T or F; this evaluation is defined recursively as follows. Given a boolean combination ψ of the boolean constants T and F, let $[\![\psi]\!]$ be the one of these constants defined in the usual way. Then we may recursively define an evaluation function for QBF formulae evalq by

$$\mathsf{evalq}(\varphi) = \begin{cases} \mathsf{evalq}(\varphi'[x_i := \mathsf{F}]) \wedge \mathsf{evalq}(\varphi'[x_i := \mathsf{T}]) & \text{if } \varphi = \forall x_i \varphi' \\ \mathsf{evalq}(\varphi'[y_i := \mathsf{F}]) \vee \mathsf{evalq}(\varphi'[y_i := \mathsf{T}]) & \text{if } \varphi = \exists y_i \varphi' \\ [\![\varphi]\!] & \text{if } \varphi \text{ is not quantified} \end{cases}$$

where $\varphi'[v := b]$ is the formula obtained by substituting the boolean constant b for all occurrences of the variable v in φ'.

Definition 14 (3-QUANTIFIED BOOLEAN FORMULA). *Given a 3QBF formula φ, the decision problem* 3-QUANTIFIED BOOLEAN FORMULA *asks if* $\mathsf{evalq}(\varphi) = \mathsf{T}$.

3-QUANTIFIED BOOLEAN FORMULA is well-known to be PSPACE-complete as was shown by Stockmeyer and Meyer [122]. Our proof will exploit the following definition and lemma about QBF formulas.

Definition 15. *An* assignment function *for a QBF formula of the form (2.5) is a function* $\mathsf{assign} : (X \to \mathbb{B}) \to (Y \to \mathbb{B})$ *such that*

1. *For all $A \in (X \to \mathbb{B})$, $(A \cup \mathsf{assign}(A))$ makes f true.*

2. *For all $i \in \{1, \ldots, n\}$ and all $A_1, A_2 : X \to \mathbb{B}$ we have that if A_1 and A_2 agree on their assignments to the variables $x_n, \ldots, x_i$, then* $\mathsf{assign}(A_1)(y_i) = \mathsf{assign}(A_2)(y_i)$

Lemma 10. *A QBF formula has an assignment function if and only if it evaluates to* T.

Proof. Easy. □

The next subsection defines the protocol used in our reduction.

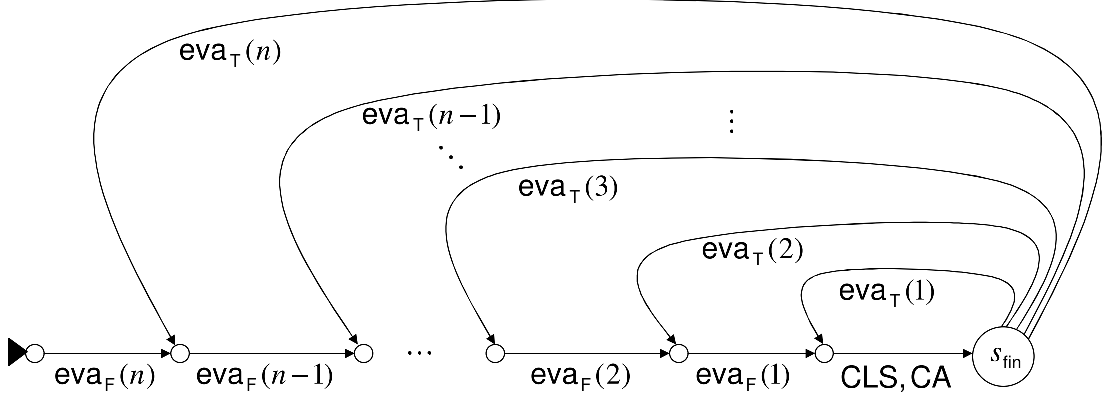


Figure 2.7: The protocol $\mathcal{P}_\varphi$

2.6.1 The Protocol $\mathcal{P}_\varphi$

Given a 3QBF formula φ of the form (2.5), we map φ to a protocol $\mathcal{P}_\varphi$ as shown in Fig. 2.7. Our encoding can be viewed as a generalization of Gibbons and Korach's reduction used to prove that checking sequential consistency of a single trace is NP-complete [66]. For simplicity we assume that each clause C_j has *exactly* three literals.[15] The various transition labels are the following traces.

- For each $1 \leq i \leq n$, and $b \in \mathbb{B}$, $\mathsf{eva}_b(i)$ is defined in Fig. 2.8. $\mathsf{eva}_b(i)$ stands for "encode variable assignments", and, intuitively, is responsible for the mechanisms that force the assignment $x_i := b$, and existentially select some value of $\mathbb{B}$ for y_i.[16]

- CLS is defined to be $\prod_{j=1}^{m} \mathsf{ec}'(j)$, where $\mathsf{ec}'(j)$ is defined in Fig. 2.9. CLS is short for "clauses" and encodes the mechanism responsible for mapping variable assignments to satisfied clauses.

- Finally, the trace CA is defined as follows.

$$\mathsf{CA} = \left\langle \prod_{j=1}^{m} (R, p_c, c_j, \mathsf{T}) \prod_{i=1}^{n} (R, p_c, \mathsf{hb}_i, 0) \right\rangle \tag{2.6}$$

CA stands for "check assignment", and intuitively is responsible for checking that the current variable assignment satisfies all clauses (and hence f).

[15] Repetition of literals in clauses can always make this so.

[16] $\mathsf{eva}_b(i)$ is a similar widget to the trace $\mathsf{eva}(i)$ used in Sect. 2.4. However, the former makes the assignment $x_i := b$ *and* existentially selects an assignment for y_i, while the latter simply existentially selects an assignment for x_i.

On an intuitive level, our reduction works as follows. If φ evaluates to true, then it has an assignment function, and we show how any assignment function can be used to construct a DSC-reordering of any trace of $\mathcal{P}_\varphi$. Roughly, each time the transition of Fig. 2.7 labeled “$\mathsf{CLS\ CA}$” is followed, there is a notion of the “current” assignment to the variables X; this current assignment is dictated by the path previously taken through the states of $\mathcal{P}_\varphi$. The assignment function maps the current assignment of X to an assignment to the existential variables Y, and produces a total assignment that satisfies all clauses, hence allowing for a DSC-reordering. Sect. 2.6.2 proves this direction of the reduction.

Conversely, if $\mathcal{P}_\varphi$ is DSC, then so too is a particular trace of $\mathcal{P}_\varphi$ that we call the *intended trace*. The intended trace corresponds to a run through $\mathcal{P}_\varphi$ such that each of the 2^n possible assignments to the universally quantified variables X is explored. We show that from any DSC-reordering of the intended trace one can extract an assignment function for φ. This direction of the reduction is argued in Sect. 2.6.3.

The fact that our reduction can be computed in polynomial time is now asserted.

Lemma 11. *For any φ, $\mathcal{P}_\varphi$ can be constructed in time polynomial in $|\varphi|$.*

Proof. Each of the traces labeling transitions in Fig. 2.7 has polynomial length. In particular, $|\mathsf{eva}_b(i)| = \mathcal{O}(m+n)$, $|\mathsf{CLS}| = \mathcal{O}(m)$, and $|\mathsf{CA}| = \mathcal{O}(m+n)$. Since $\mathsf{eva}_b(i)$ occurs $2n$ times, and CLS and CA both occur once, $\mathcal{P}_\varphi$ has size $\mathcal{O}(|\phi|^2)$ and the result follows. □

We now discuss the processors and addresses used in $\mathcal{P}_\varphi$ and provide intuition for what role they play.

Addresses

Here we outline the various addresses used. The intuition attached should be read in the context of some DSC-reordering π of a trace of $\mathcal{L}(\mathcal{P}_\varphi)$.

- For each $v_i \in X \cup Y$, there is an address v_i. v_i can hold one of the values T, F, or X, which respectively denote the variable v_i being assigned T, assigned F, or unassigned. The unassigned value X is used to prevent “old” assignments to persist past certain points in a reordering. At any point in a reordering, there is the notion of a *current variable assignment*, which corresponds to the value written by the most recent write to each v_i.

p_c	p_i	$x_i^{\mathsf{T},1}$	$x_i^{\mathsf{T},2}$	$x_i^{\mathsf{F},1}$	$x_i^{\mathsf{F},2}$	$y_i^{\mathsf{T},1}$	$y_i^{\mathsf{T},2}$	$y_i^{\mathsf{F},1}$	$y_i^{\mathsf{F},2}$
$\left\langle \prod_{j=1}^{i} (R, \mathsf{hb}_j, 2) \right\rangle \dagger$									
	$\left\langle \begin{array}{c} (W, x_i, \mathsf{X})(W, y_i, \mathsf{X}) \\ \prod_{j=1}^{m} (W, c_j, \mathsf{X}) \\ (W, \mathsf{hb}_i, 0) \end{array} \right\rangle$								
	$(W, \mathsf{hb}_i, 1)$								
	$(W, \mathsf{hb}_i, 2)$								
		$(R, \mathsf{hb}_i, 0)$							
		(W, x_i, T)							
		$(R, \mathsf{hb}_i, 0)\dagger$							
		$(R, \mathsf{hb}_i, 1)$							
				$(R, \mathsf{hb}_i, 0)$					
				(W, x_i, F)					
				$(R, \mathsf{hb}_i, 0)\ddagger$					
				$(R, \mathsf{hb}_i, 1)$					
			$(R, \mathsf{hb}_i, 0)$						
			$(R, \mathsf{hb}_i, 1)$						
			(R, x_i, T)						
					$(R, \mathsf{hb}_i, 0)$				
					$(R, \mathsf{hb}_i, 1)$				
					(R, x_i, F)				
						$(R, \mathsf{hb}_i, 0)$			
						(W, y_i, T)			
						$(R, \mathsf{hb}_i, 1)$			
								$(R, \mathsf{hb}_i, 0)$	
								(W, y_i, F)	
								$(R, \mathsf{hb}_i, 1)$	
							$(R, \mathsf{hb}_i, 0)$		
							$(R, \mathsf{hb}_i, 1)$		
							(R, y_i, T)		
									$(R, \mathsf{hb}_i, 0)$
									$(R, \mathsf{hb}_i, 1)$
									(R, y_i, F)

Figure 2.8: The trace $\mathsf{eva}_b(i)$, where $b \in \mathbb{B}$ and $i \in 1, \ldots, n$. The trace is listed vertically, i.e. events higher on the page precede lower events. If $b = \mathsf{T}$, the events marked † are included while the event marked ‡ is not; if $b = \mathsf{F}$, the event ‡ is included while those marked with † are not. Note the processor names are omitted from events; all events in each column are performed by the processor heading the column.

u^j	$\langle (R, u, \mathsf{pol}_j(u))\ (W, c_j, \mathsf{T}) \rangle$		
v^j		$\langle (R, v, \mathsf{pol}_j(v))\ (W, c_j, \mathsf{T}) \rangle$	
w^j			$\langle (R, w, \mathsf{pol}_j(w))\ (W, c_j, \mathsf{T}) \rangle$

Figure 2.9: The trace $\mathsf{ec}'(j)$. Here the variables of C_j are $u, v, w \in X \cup Y$, and $\mathsf{pol}_j(u)$ (resp. $\mathsf{pol}_j(v)$, $\mathsf{pol}_j(w)$) is the polarity of the literal of u (resp. v, w) in C_j, i.e. T or F. The events of each row are all done by the processor named on the left, hence processor names are omitted from the events. We note that $\mathsf{ec}'(j)$ is the same (modulo variable renaming) as the trace $\mathsf{eva}_j()$ of Fig. 2.4 employed in Sect. 2.4 and the two traces serve similar roles in their respective reductions.

- For each $i \in \{1, \ldots, n\}$, there is a *heartbeat* address hb_i. In any reordering, these cycle through the values 0, 1, and 2. The phases of hb_i play an important role in our reasoning about reordering.

- For each $1 \leq j \leq m$, there is an address c_j. This address can hold the values T or X. c_j will only store T if the current variable assignment satisfies the clause C_j. When X is written to c_j, this serves to prevent future events from reading a previous T.

Processors

$\mathcal{P}_\varphi$ involves the following five classes of processors:

- A single processor named p_c. This processor performs all events in CA, and a few events in each $\mathsf{eva}_b(i)$.

- For each $i \in \{1, \ldots, n\}$, a processor p_i. Processor p_i is the only writer to address hb_i. This ensures that in any reordering, the sequence of values in address hb_i is $0, 1, 2, 0, 1, 2, \ldots$.

- For each $i \in \{1, \ldots, n\}$ four processors named $x_i^{\mathsf{T},1}$, $x_i^{\mathsf{T},2}$, $x_i^{\mathsf{F},1}$, and $x_i^{\mathsf{F},2}$. These are the processors involved in the mechanism that performs an assignment to the variable x_i.

- For each $i \in \{1, \ldots, n\}$, four processors named $y_i^{\mathsf{T},1}$, $y_i^{\mathsf{T},2}$, $y_i^{\mathsf{F},1}$, and $y_i^{\mathsf{F},2}$. These are analogous to the previous item.

- For each $j \in \{1, \ldots, m\}$ such that variable x_i (y_i) appears in clause C_j, a processor named x_i^j (y_i^j). These are the processors involved in CLS; see Fig. 2.9. Intuitively, processor v_i^j tries to read the polarity that clause C_j has for variable v_i(either T or F) from the current assignment to v_i. If it can, then it indicates that clause C_j is satisfied by writing T to address c_j.

2.6.2 Proof that $\mathcal{P}_\varphi$ is DSC Implies φ

In this section, we assume that $\mathcal{P}_\varphi \in$ DSC, and show that this implies φ evaluates to T. A high-level view of the proof is as follows. We identify a specific trace $\mathsf{it}_\varphi \in \mathcal{L}(\mathcal{P}_\varphi)$ that we call the intended trace.[17] Since it_φ is DSC, it has at least one DSC reordering. Most of this subsection is devoted to arguing that from *any* DSC reordering π of it_φ, one can extract an assignment function for φ, hence by Lemma 10, φ is true. Rigor is paramount here; we must ensure that it_φ doesn't have any reorderings other than those from which we can extract an assignment function.

Definition 16 (intended trace it_φ). *For $i \in \{1, \ldots, n\}$, define the trace it^i_φ recursively as follows*

$$\mathsf{it}^i_\varphi = \begin{cases} \mathsf{CLS}\ \mathsf{CA} & \text{if } i = 0 \\ \mathsf{eva}_\mathsf{F}(i)\ \mathsf{it}^{i-1}_\varphi\ \mathsf{eva}_\mathsf{T}(i)\ \mathsf{it}^{i-1}_\varphi & \text{if } i > 0 \end{cases} \tag{2.7}$$

We define it_φ to be it^n_φ.

Lemma 12. $\mathsf{it}_\varphi \in \mathcal{L}(\mathcal{P}_\varphi)$.

Proof. Follows easily from the definitions of it_φ and $\mathcal{P}_\varphi$. □

For the remainder, let π be an arbitrary DSC-reordering of it_φ.

Extending our *phase* terminology introduced in section 2.2, we say that o *is in the most recent r-phase of* hb_i if

1. o is in a r-phase of hb_i, and
2. the occurrence of the last write of r to address hb_i before o in $\mathsf{it}_\varphi{}^\pi$ is also the the last write of r to hb_i before o in it_φ.

Definition 17 (sees). *For an atomicity construct σ in it_φ and an address a, we say that σ* sees *v* for *a if in $\mathsf{it}_\varphi{}^\pi$, the last write to a before σ writes value v.*

[17] The reason it is the *intended* trace is that ideally we would map φ to a protocol with it_φ and its prefixes being the only traces. However, since the length of it_φ is exponential in $|\varphi|$, this would not constitute a polynomial time reduction. Hence we must "role-up" it_φ into the automaton $\mathcal{P}_\varphi$, which only has a polynomial number of states. Because "rolling-up" entails cycles, $\mathcal{P}_\varphi$ has many traces other than the intended trace and its prefixes; these superfluous traces add to the complexity of the proof of the converse implication of Sect. 2.6.3.

Definition 18 (observes). *For an occurrence o of* CA*, and some $A_X : X \to \mathbb{B}$, we say that o* observes A_X *if for each $x_i \in X$, o either sees $A_X(x_i)$ for x_i or sees* X *for x_i. The term is used in an analogous sense for assignments in $Y \to \mathbb{B}$.*

Definition 19 (consistent observation). *Let $0 \le \ell \le \ell' < 2^n$, and let $y_i \in Y$. We say that the $[\ell, \ell']$-occurrences of* CA *share*[18] *a* consistent observation *of y_i if there are no $k, k' \in [\ell, \ell']$ such that the kth and k'th occurrences of* CA *respectively see* T *and* F *for y_i.*

We now present the main theorem, followed by its proof and several supporting lemmas.

Theorem 9. *If $\mathsf{it}_\varphi \in$ DSC then $\mathsf{evalq}(\varphi) = \mathsf{T}$.*

Proof. Given a DSC-reordering π of it_φ, we show how to construct an assignment function for φ; the result then follows from Lemma 10. If we identify any X-assignment A_X with the binary number $A_X(x_n)A_X(x_{n-1})\ldots A_X(x_1)$, then from Lemma 15 we will see that the A_Xth occurrence of CA observes A_X; we refer to this occurrence as the A_X-occurrence. For each $i \in \{1, \ldots, n\}$, let k_i be the unique natural such that $k_i 2^{i-1} \le A_X \le (k_i + 1)2^{i-1} - 1$. From Lemma 17 we will find that the $k_i 2^{i-1}$ through $(k_i + 1)2^{i-1} - 1$ occurrences of CA share a consistent observation of y_i. Now, define the function $\mathsf{assign} : (X \to \mathbb{B}) \to (Y \to \mathbb{B})$ such that $\mathsf{assign}(A_X)(y_i)$ is T (F) if any of these occurrences of CA see T (F) for y_i, and set $\mathsf{assign}(A_X)(y_i) = \mathsf{F}$ if[19] all occurrences see X for y_i. That assign is well-defined follows from Lemma 17. To see that assign is an assignment function, we note that Lemma 19 will ensure that condition 1 of Def. 15 is satisfied, i.e. that for any A_X, $\mathsf{assign}(A_X) \cup A_X$ satisfies f. Also, condition 2 is clearly satisfied by our definition of assign. □

Lemma 13. *For any $v_i \in X \cup Y$, there can be at most one write to v_i in any 0-phase of hb_i, and the value written is in $\mathbb{B}$.*

Proof. Let H_0 be a 0-phase of hb_i, and let H_1 be the immediately next phase of hb_i, which is necessarily a 1-phase, by (2.8) below. Assume $v_i = y_i$; the case $v_i = x_i$ is similar. Observe that

$$\begin{aligned} \mathsf{it}_\varphi \uparrow (W, *, \mathsf{hb}_i, *) &= ((W, p_i, \mathsf{hb}_i, 0)(W, p_i, \mathsf{hb}_i, 1)(W, p_i, \mathsf{hb}_i, 2))^{2^{n-i+1}} \\ &= {\mathsf{it}_\varphi}^\pi \uparrow (W, *, \mathsf{hb}_i, *) \end{aligned} \tag{2.8}$$

[18] where the occurrences of CA are numbered starting from 0.
[19] The choice of F here is arbitrary; T would work too.

Note that any write of X to v_i occurs in an atomicity construct in the 2nd column of Fig. 2.8. Because of this atomicity construct and (2.8), these writes always occur in a 2-phase of hb_i.[20] Therefore any write to v_i in H_0 must write a value in $\mathbb{B}$.

Now the events in the 7th and 9th columns of Fig. 2.8 ensure all writes in $(W, *, y_i, \mathbb{B})$ are in either the most recent 0-phase or the most recent 1-phase of hb_i. This follows from the fact that for each $b \in \mathbb{B}$,

$$\begin{aligned} \mathsf{it}_\varphi \uparrow (*, y_i^{b,1}, *, *) &= ((R, y_i^{b,1}, \mathsf{hb}_i, 0)(W, y_i^{b,1}, y_i, b)(R, y_i^{b,1}, \mathsf{hb}_i, 1))^{2^{n-i+1}} \\ &= \mathsf{it}_\varphi{}^\pi \uparrow (*, y_i^{b,1}, *, *) \end{aligned} \tag{2.9}$$

Hence the inheritance relation induces a one-to-one correspondence between the reads of 0 (resp. reads of 1) from hb_i in (2.9) and the writes of 0 (resp. writes of 1) to hb_i in (2.8). Because of the "sandwiching" of the writes in (2.9), exactly one occurrence of each of $(W, y_i^{\mathsf{F},1}, y_i, \mathsf{F})$ and $(W, y_i^{\mathsf{T},1}, y_i, \mathsf{T})$ occurs in $H_0 \cup H_1$. However, in a similar manner, the events of the 8th and 10th columns of Fig. 2.8 enforce that the reads of y_i in these columns occur in the most recent 1-phase of hb_i. Thus, if H_0 contains an occurrence of *both* writes $(W, y_i^{\mathsf{F},1}, y_i, \mathsf{F})$ and $(W, y_i^{\mathsf{T},1}, y_i, \mathsf{T})$, one of the reads of y_i in columns 8 and 10 will fail to inherit the correct value. Hence H_0 contains at most one of these writes. □

Lemma 14. *Each occurrence of* CA *in* $\mathsf{it}_\varphi{}^\pi$ *is in the most recent 0-phase of* hb_i*, for all* $i \in \{1, \ldots, n\}$.

Proof. We first note that for each $i \in \{1, \ldots, n\}$, every occurrence of CA is in *some* 0-phase of hb_i; this follows from the atomicity of CA (2.6) and its read of 0 from hb_i. Hence we must argue that it is always the most recent 0-phase.

There are 2^n occurrences of CA in $\mathsf{it}_\varphi{}^\pi$. We index these from left to right by n-bit binary numbers, starting at all-zeros, and counting in binary up to all-ones. We prove the lemma by induction on the index. For the $\mathsf{it}_\varphi{}^\pi$-leftmost (i.e. all-zeros) occurrence, the lemma holds trivially, since there is only a single prior occurrence of $(W, p_i, \mathsf{hb}_i, 0)$. For the inductive step, let $\ell = \ell_n, \ldots, \ell_1$ be the index of an occurrence such that $\ell > 0$ and the lemma holds for index $\ell - 1$. Let k be minimal such that $\ell_k = 1$. From Def. 16, in-between occurrence $\ell - 1$ and ℓ (in it_φ) is precisely the trace

$$\mathsf{eva}_\mathsf{T}(k) \prod_{i=1}^{k-1} \mathsf{eva}_\mathsf{F}(k-i) \tag{2.10}$$

[20] This is not quite true; the very first write of X to v_i will occur before hb_i is assigned.

It follows that for each $i \in \{k+1, \ldots, n\}$, the ℓth occurrence of CA is certainly in the most recent 0-phase of hb_i, since (by IH) this is true of the $(\ell-1)$th occurrence, and, in it_φ, there are no writes to hb_i between the two occurrences.

For the case of $i \in \{1, \ldots, k\}$, we note that a prefix of (2.10) is

$$\left\langle \prod_{j=1}^{k} (R, p_c, \mathsf{hb}_j, 2) \right\rangle \tag{2.11}$$

This atomicity construct must occur in $\mathsf{it}_\varphi{}^\pi$ in the most recent 2-phase of each of $\mathsf{hb}_1, \ldots, \mathsf{hb}_k$. This follows from the fact that the processor of (2.11) is p_c, which is also the processor of the $(\ell-1)$th occurrence of CA, which (by IH) is in the most recent 0-phases of these addresses. Now we note that in it_φ there is exactly one write of value 0 to each of $\mathsf{hb}_1, \ldots, \mathsf{hb}_k$ between the $(\ell-1)$th and the ℓth occurrence of CA (i.e. the writes that occur in (2.10). But since these occur after (2.11), these are the only writes of 0 to each of $\mathsf{hb}_1, \ldots, \mathsf{hb}_k$ that could possibly impart values to the ℓth occurrence of CA. Of importance here is the fact that all writes to any hb_i are performed by the same processor, p_i. Therefore, for $i \in \{1, \ldots, k\}$, the ℓth occurrence of CA is in the most recent 0-phase of hb_i. ☐

Lemma 15. *Let $\ell = \ell_n \ldots \ell_1$ be a binary number.*[21] *Then the ℓth occurrence of* CA *observes the assignment A_X defined by $A_X(x_i) = \ell_i$ for all $i \in \{1, \ldots, n\}$.*

Proof. Let o be the ℓth occurrence of CA. We show that for any $i \in \{1, \ldots, n\}$, o either sees ℓ_i or X for x_i; the lemma follows. From Lemma 14, o occurs in the most recent 0-phase of hb_i; call this phase H_0. Immediately preceding H_0 is a 2-phase of hb_i which we call H_2, and immediately superseding H_0 is a 1-phase we call H_1. From the structure of it_φ, the most recent instance of $\mathsf{eva}_b(i)$ prior to o in it_φ has $b = \ell_i$; for the rest of this proof, $\mathsf{eva}_b(i)$ refers to this specific instance of the trace. The event $(W, p_i, \mathsf{hb}_i, 0)$ in $\mathsf{eva}_b(i)$ is the first event of H_0. By Lemma 16, for each $r \in \{0, 1\}$, all events of $(R, *, \mathsf{hb}_i, r)$ in $\mathsf{eva}_b(i)$ are in H_r. From Lemma 13, there is at most one write to x_i in H_0, and the value written is in $\mathbb{B}$. Now if $b = \mathsf{T}$, then the event marked † in Fig. 2.8 is present, thus forcing $(W, x_i^{\mathsf{T},1}, x_i, \mathsf{T})$ to be in H_0. Similarly, if $b = \mathsf{F}$, then the event marked ‡ forces $(W, x_i^{\mathsf{F},1}, x_i, \mathsf{F})$ to be in H_0. In either case, there is exactly one write w to x_i in H_0, and the value written is b. Now, if o sees w, then we are done. Otherwise, it must be that in $\mathsf{it}_\varphi{}^\pi$, o is prior

[21] It is convenient here to index the bits of ℓ such that ℓ_1 (rather than the traditional ℓ_0) is the lowest order bit

to w. In this case, since o is in H_0, the only write to x_i that o can see is the event $(W, p_i, x_i, \mathsf{X})$ performed in H_2, immediately prior to entering H_0 (see column for p_i in Fig. 2.8). Therefore A_X is observed. □

Lemma 16. *For any $b \in \mathbb{B}$, $r \in \{0, 1\}$, $i \in \{1, \ldots, n\}$, and occurrence o of $\mathsf{eva}_b(i)$ in it_φ, all $(R, *, \mathsf{hb}_i, r)$ events in o are in their most recent r-phase of hb_i.*

Proof. In o, any $(R, *, \mathsf{hb}_i, r)$ event is performed by a processor in the set $P = \left\{x_i^{\mathsf{T},1}, x_i^{\mathsf{T},2}, x_i^{\mathsf{F},1}, x_i^{\mathsf{F},2}, y_i^{\mathsf{T},1}, y_i^{\mathsf{T},2}, y_i^{\mathsf{F},1}, y_i^{\mathsf{F},2}\right\}$. Note that for any $p \in P$, by Fig. 2.8 we have either

$$\mathsf{it}_\varphi \upharpoonright (R, p, \mathsf{hb}_i, *) \;=\; ((R, p, \mathsf{hb}_i, 0)(R, p, \mathsf{hb}_i, 0)(R, p, \mathsf{hb}_i, 1))^{2^{n-i+1}}$$

or

$$\mathsf{it}_\varphi \upharpoonright (R, p, \mathsf{hb}_i, *) \;=\; ((R, p, \mathsf{hb}_i, 0)(R, p, \mathsf{hb}_i, 1))^{2^{n-i+1}}$$

and in both cases

$$\mathsf{it}_\varphi \upharpoonright (R, p, \mathsf{hb}_i, *) \;=\; \mathsf{it}_\varphi{}^\pi \upharpoonright (R, p, \mathsf{hb}_i, *)$$

since all events in the projection are performed by the same processor. The lemma follows by (2.8). □

Lemma 17. *For any $i \in \{1, \ldots, n\}$, and natural number k such that*

$$0 \;\leq\; k2^{i-1} \;\leq\; (k+1)2^{i-1} - 1 \;\leq\; 2^n - 1$$

the $[k2^{i-1}, (k+1)2^{i-1} - 1]$-occurrences of CA share a consistent observation of y_i.

Proof. From Lemma 14, each of the $[k2^{i-1}, (k+1)2^{i-1} - 1]$-occurrences of CA are in their most recent 0-phases of hb_i. Furthermore, from Def. 16, these occurrences must therefore be in *the same* 0-phase of hb_i; call this phase H_0. From Lemma 13, there can be at most one write to y_i in H_0, and this write writes some value $b \in \mathbb{B}$. Also, the most recent write to y_i prior to H_0 writes value X; see the second column of Fig. 2.8. Thus each of the $[k2^{i-1}, (k+1)2^{i-1} - 1]$-occurrences of CA see either X or b for y_i. □

Lemma 18. *For each event $(R, p_c, c_j, \mathsf{T})$, there exists an event $(W, v_i^{b,1}, v_i, b)$, where $b \in \mathbb{B}$ such that*

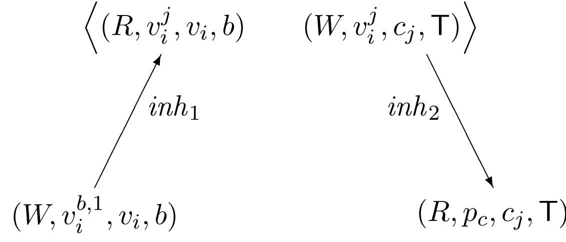


Figure 2.10: The inheritance edges discussed in the proof of Lemma 18

1. *the read event sees b for v_i, and*

2. *v_i appears in C_j in polarity b.*

Proof. Throughout the proof, *the read* (resp. *the write*) will refer to the events in the lemma statement. The read must inherit from some write $(W, v_j^i, c_j, \mathsf{T})$, which is atomically coupled with some read (R, v_j^i, v_i, b), for some $b \in \mathbb{B}$, which inherits from the write. We let a refer to this two-event atomic subtrace. See Fig. 2.10 for an illustration, along with labellings of the two inheritance edges involved.

We first argue that the write and the read are in the same 0-phase of hb_i. From Lemma 14, the read is in its most recent 0-phase of hb_i; call this phase H_0. Now suppose the write is in some phase of hb_i that is prior to H_0. Consider the inheritance edges inh_1 and inh_2 of Fig. 2.10. If a occurs prior to H_0, then inh_2 cannot hold, while if a occurs in H_0, then inh_1 cannot hold; these facts follow from the writes atomically coupled with $(W, p_i, \mathsf{hb}_i, 0)$ (see second column of Fig. 2.8). Hence we conclude that the write is in H_0 also.

From Lemma 13, the write is the only write to v_i in H_0, and clearly because of inh_1 and inh_2, the write must occur prior to the read in H_0. Therefore, the read sees b for v_i.

The fact that v_i appears in C_j in polarity b follows from the definition of $\mathsf{ec}'(j)$ in Fig. 2.9. $\square$

Lemma 19. *Let $A : (X \cup Y) \to \mathbb{B}$ be an assignment that is observed by some occurrence of* CA. *Then A satisfies f.*

Proof. We argue that an arbitrary clause C_j is satisfied by A. Let o be the occurrence of $(R, p_c, c_j, \mathsf{T})$ in the occurrence of CA that observes A. From Lemma 18, there exists an event

$(W, v_i^{b,1}, v_i, b)$, where $b \in \mathbb{B}$ such that conditions 1 and 2 of Lemma 18 hold. Since condition 1 holds, we have that $A(v_i) = b$. Since condition 2 holds, C_j is satisfied when $v_i = b$. □

2.6.3 Proof that φ Implies $\mathcal{P}_\varphi$ is DSC

For this section, we assume that φ is true, and fix an assignment function assign for φ, which exists thanks to Lemma 10. Let $K_\varphi \subset \mathcal{L}(\mathcal{P}_\varphi)$ be the set of traces that leave $\mathcal{P}_\varphi$ in state s_{fin}. First, we note that any trace of $\mathcal{L}(\mathcal{P}_\varphi)$ is a prefix of some trace of K_φ. Since DSC is a prefix-closed property, we thus need only consider traces in K_φ; i.e. K_φ DSC implies $\mathcal{P}_\varphi \in$ DSC. Any trace $\sigma \in K_\varphi$ uniquely defines a run (i.e. sequence of states) of $\mathcal{P}_\varphi$; we let $\#(\sigma)$ be the number of occurrences of s_{fin} in this run. We will prove that for all $\sigma \in K_\varphi$, σ is DSC, by induction on $\#(\sigma)$. In fact, we will prove the stronger assertion that all traces of K_φ have a DSC reordering with certain properties; these properties are defined subsequently in Def. 21. Throughout, we will identify a reordering π of a trace τ with τ^π.

Definition 20 (final assignment). *Suppose $\sigma \in K_\varphi$, has DSC-reordering π. Then the* final X-assignment *of π is the function $\mathsf{fa}_X^\pi : X \to (\mathbb{B} \cup \{\mathsf{X}\})$ defined as follows. For each $i \in \{1, \dots, n\}$, $\mathsf{fa}_X^\pi(x_i)$ is the value written by the π-most recent event of $(W, *, x_i, *)$ prior to the last occurrence of CA in π. The* final Y-assignment fa_Y^π *is defined analogously. If both $\mathsf{X} \notin \mathsf{fa}_X^\pi(X)$ and $\mathsf{X} \notin \mathsf{fa}_Y^\pi(Y)$, we say that the final assignments of π are* well defined.

Definition 21 (preferred reordering). *A reordering π of $\sigma \in K_\varphi$ is called a* preferred reordering *if the following all hold.*

1. *π is a DSC-reordering of σ*
2. *A trace of the form*

$$\mathsf{CA} \prod_{i=1}^{n} (W, p_i, \mathsf{hb}_i, 1)\ \alpha_i\ \beta_i\ (W, p_i, \mathsf{hb}_i, 2) \tag{2.12}$$

   is a suffix of π, where

   - *α_i includes only events of processors $x_i^{\mathsf{T},1}, x_i^{\mathsf{T},2}, x_i^{\mathsf{F},1}, x_i^{\mathsf{F},2}, y_i^{\mathsf{T},1}, y_i^{\mathsf{T},2}, y_i^{\mathsf{F},1}, y_i^{\mathsf{F},2}$, and*
   - *β_i includes only events of processors $x_i^1, y_i^1, \dots, x_i^m, y_i^m$, and all reads of β_i inherit from events in α_i.*

3. *The final assignments of* π *are well-defined, and furthermore* $\mathsf{assign}(\mathsf{fa}^{\pi}_{X}) = \mathsf{fa}^{\pi}_{Y}$.

Definition 22. *Let* A *be an assignment that interprets the variables of* Y.[22] *For each* $i \in \{1, \ldots, n\}$ *and* $b \in \mathbb{B}$, *the traces* $\mathsf{eva}_b(i)^A_0$ *and* $\mathsf{eva}_b(i)^A_1$ *are defined in Fig. 2.11.*

We now state and prove the main result of this section.

Theorem 10. *For all* $\sigma \in K_\varphi$, σ *has a preferred reordering.*

Proof. By induction on $\#(\sigma)$. **Base Case.** There is a unique trace $\sigma_1 \in K_\varphi$ such that $\#(\sigma_1) = 1$, namely

$$\sigma_1 = \mathsf{eva_F}(n)\ \mathsf{eva_F}(n-1)\ldots\mathsf{eva_F}(1)\ \mathsf{CLS}\ \mathsf{CA}$$

We argue that σ_1 has a preferred reordering. Let $A_\mathsf{F} : X \to \mathbb{B}$ be the function that is constantly F, and let $A = \mathsf{assign}(A_\mathsf{F}) \cup A_\mathsf{F}$. Then π is defined by

$$\begin{aligned} \pi \quad = \quad & \textstyle\prod_{i=1}^{n} \left\langle (W, x_i, \mathsf{X})(W, y_i, \mathsf{X}) \left(\prod_{j=1}^{m}(W, c_j, \mathsf{X})\right)(W, \mathsf{hb}_i, 0)\right\rangle_{p_i} \\ & \textstyle\prod_{i=1}^{n} \mathsf{eva_F}(i)^A_0 \\ & \mathsf{CLS}_0 \\ & \mathsf{CA} \\ & \textstyle\prod_{i=1}^{n} (W, p_i, \mathsf{hb}_i, 1)\mathsf{eva_F}(i)^A_1 \mathsf{CLS}(i)_1 (W, p_i, \mathsf{hb}_i, 2) \end{aligned}$$

We leave it as an exercise for the reader to verify that there exists a shuffling of CLS into the $n+1$ traces $\mathsf{CLS}_0, \mathsf{CLS}(1)_1, \ldots, \mathsf{CLS}(n)_1$ such that π is a preferred reordering of σ_1; for a hint on how this is done, see the analogous traces in the inductive step.

Inductive Step. Let $\sigma \in K$ be such that $\#(\sigma) > 1$. Then σ can be factored as $\sigma = \sigma'\sigma''$, where $\sigma' \in K$, $\#(\sigma') = \#(\sigma) - 1$, and σ' has a preferred reordering π'. Furthermore, from the structure of $\mathcal{P}_\varphi$, we have that

$$\sigma'' = \mathsf{eva_T}(k)\ \mathsf{eva_F}(k-1)\ \mathsf{eva_F}(k-2)\ \ldots\ \mathsf{eva_F}(1)\ \mathsf{CLS}\ \mathsf{CA}$$

for some $k \in \{1, \ldots, n\}$. We will construct a preferred reordering π of σ by inserting the events of σ'' into π'. Let $A_X : X \to \mathbb{B}$ be defined by

$$A_X(x_i) = \begin{cases} \mathsf{fa}^{\pi'}_X(x_i) & \text{if } k < i \le n \\ \mathsf{T} & \text{if } i = k \\ \mathsf{F} & \text{if } 1 \le i < k \end{cases} \tag{2.13}$$

[22] In other words, Y is contained in the domain of A.

We will define π such that the final X-assignment of π is A_X. Factor $\pi' = \pi'_1\pi'_2$ where

$$\pi'_2 \quad = \quad \prod_{i=k+1}^{n} (\ (W, p_i, \mathsf{hb}_i, 1)\ \alpha_i\ \beta_i\ (W, p_i, \mathsf{hb}_i, 2)\ ) \tag{2.14}$$

That this factoring can be done follows from the inductive hypothesis.

We now shuffle the events of CLS into $n+1$ traces $\mathsf{CLS}_0, \mathsf{CLS}(1)_1, \ldots, \mathsf{CLS}(n)_1$. Let $A = \mathsf{assign}(A_X) \cup A_X$. CLS involves $3m$ atomicity constructs (each of two events). Each of these atomicity constructs is performed by a processor v_i^j, where variable v_i appears in clause C_j. Let CLS_0 be a trace (order is irrelevant) of all such atomicity constructs performed by any v_i^j such that variable v_i appears in clause C_j in polarity $A(v_i)$, i.e. clause C_j is satisfied by the assignment A. Also, for each $i \in \{1, \ldots, n\}$, let $\mathsf{CLS}(i)_1$ be a trace (order is irrelevant) of all atomicity constructs performed by v_i^j, where variable v appears in clause C_j in polarity $\neg A(v_i)$. Note that since assign is an assignment function, we have that for each $j \in \{1, \ldots, m\}$, CLS_0 contains at least one write of T to c_j (and no other values to c_j are written in CLS_0).

We now define π. To assist comprehension, events of π' are underlined, while events of σ'' are not underlined.

$$\begin{aligned}
\pi \quad = \quad & \underline{\pi'_1} \\
& \left\langle \textstyle\prod_{i=1}^{k} (R, p_c, \mathsf{hb}_i, 2) \right\rangle \\
& \textstyle\prod_{i=1}^{k} \left\langle (W, x_i, \mathsf{X})(W, y_i, \mathsf{X}) \left(\prod_{j=1}^{m} (W, c_j, \mathsf{X}) \right) (W, \mathsf{hb}_i, 0) \right\rangle_{p_i} \\
& \textstyle\prod_{i=1}^{k-1} \mathsf{eva}_{\mathsf{F}}(i)_0^A \\
& \mathsf{eva}_{\mathsf{T}}(k)_0^A \\
& \mathsf{CLS}_0 \\
& \mathsf{CA} \\
& \textstyle\prod_{i=1}^{k-1} (W, p_i, \mathsf{hb}_i, 1) \mathsf{eva}_{\mathsf{F}}(i)_1^A \mathsf{CLS}(i)_1 (W, p_i, \mathsf{hb}_i, 2) \\
& (W, p_k, \mathsf{hb}_k, 1) \mathsf{eva}_{\mathsf{T}}(k)_1^A \mathsf{CLS}(k)_1 (W, p_k, \mathsf{hb}_k, 2) \\
& \textstyle\prod_{i=k+1}^{n} \underline{(W, p_i, \mathsf{hb}_i, 1)}\ \underline{\alpha_i}\ \underline{\beta_i}\ \mathsf{CLS}(i)_1\ \underline{(W, p_i, \mathsf{hb}_i, 2)}
\end{aligned} \tag{2.15}$$

π can easily be seen to satisfy condition 2 of Def. 21, and the inductive hypothesis along with the structure of π imply that condition 3 is satisfied. That π is a DSC-reordering (condition 1 of Def. 21) can be proved by a detailed case analysis, which we leave to the reader. However, the following "hints" are relevant:

- π'_1 has a suffix of the form:

$$\mathsf{CA} \prod_{i=1}^{k-1} (\ (W, p_i, \mathsf{hb}_i, 1)\ \alpha_i\ \beta_i\ (W, p_i, \mathsf{hb}_i, 2)\ )$$

- For each $j \in \{1, \dots, m\}$, the read of T from c_j in the last occurrence of CA in π inherits from a write in CLS_0.
- For each $v_i \in X \cup Y$, the reads of v_i in CLS_0 inherit from events of σ'' (red events) if $1 \leq i \leq k$, and from events of π' if $k < i \leq n$.

□

2.7 Undecidability of Prefix-Closed SC

In this section, we show that SC, under our prefix-closed protocol semantics, is undecidable. The proof is via a nontrivial modification of Alur et al.'s proof that SC is undecidable for arbitrary (i.e. not necessarily prefix-closed) finite automata [8]. Note that our reduction depends heavily on SC's ability to radically change witness reorderings at the drop of a hat. Therefore it is unlikely that this proof could be adapted to handle the open problem pertaining to the (un)decidability of DSC. We reduce an undecidable problem about counter-machines, which are defined now.

Definition 23 (n-counter machine). *For $n \geq 1$, an n-*counter machine *is a finite automaton over the alphabet $\Sigma_n = \bigcup_{i=1}^{n} \{\mathsf{I}_i, \mathsf{D}_i, \mathsf{Z}_i\}$.*

An n-counter machine can be thought of as a finite automaton endowed with n unbounded (integer) counters. An occurrence of I_i (resp. D_i) increments (resp. decrements) counter i, while the symbol Z_i tests if the ith counter is zero. In the "counter-endowed" semantics, a Z_i transition may only be taken if counter i stores zero.

Definition 24 (admitted string). *$\sigma \in \Sigma_n^*$ is said to be* admitted *if, for each $1 \leq i \leq n$ and each prefix σ' of σ such that $\sigma' = \sigma''\mathsf{Z}_i$ for some σ'', we have that $|\sigma' \uparrow \{\mathsf{I}_i\}| = |\sigma' \uparrow \{\mathsf{D}_i\}|$.*

Therefore, if we view a string $\sigma \in \Sigma_n^*$ as a sequence of increment, decrement, and test-for-zero operations on n counters, σ is admitted if and only if every occurrence of a test-for-zero Z_i occurs when counter i stores value 0. Hence, the term *admitted* means legality under the counter-endowed semantics.

$x_i^{\mathsf{T},1}$	$x_i^{\mathsf{T},2}$	$x_i^{\mathsf{F},1}$	$x_i^{\mathsf{F},2}$	$y_i^{a,1}$	$y_i^{a,2}$	$y_i^{\neg a,1}$	$y_i^{\neg a,2}$
		$(R, \mathsf{hb}_i, 0)$					
		(W, x_i, F)					
		$(R, \mathsf{hb}_i, 0)$					
$(R, \mathsf{hb}_i, 0)$							
	$(R, \mathsf{hb}_i, 0)$						
			$(R, \mathsf{hb}_i, 0)$				
				$(R, \mathsf{hb}_i, 0)$			
				(W, y_i, a)			
						$(R, \mathsf{hb}_i, 0)$	
					$(R, \mathsf{hb}_i, 0)$		
							$(R, \mathsf{hb}_i, 0)$
			$(R, \mathsf{hb}_i, 1)$				
			(R, x_i, F)				
(W, x_i, T)							
$(R, \mathsf{hb}_i, 1)$							
		$(R, \mathsf{hb}_i, 1)$					
	$(R, \mathsf{hb}_i, 1)$						
	(R, x_i, T)						
				$(R, \mathsf{hb}_i, 1)$			
					$(R, \mathsf{hb}_i, 1)$		
					(R, y_i, a)		
						$(W, y_i, \neg a)$	
						$(R, \mathsf{hb}_i, 1)$	
							$(R, \mathsf{hb}_i, 1)$
							$(R, y_i, \neg a)$

Figure 2.11: The trace $\mathsf{eva}_{\mathsf{F}}(i)_0^A$ (top half of table) and $\mathsf{eva}_{\mathsf{F}}(i)_1^A$ (bottom half of table). Here, $a = A(y_i)$ is the value assigned to the existentially quantified variable y_i. Note that these two traces can be seen as being obtained by permuting $\mathsf{eva}_{\mathsf{F}}(i)$ while preserving per-processor order, and then factoring the result into two subtraces. The traces $\mathsf{eva}_{\mathsf{T}}(i)_0^A$ and $\mathsf{eva}_{\mathsf{T}}(i)_1^A$ are defined similarly; in particular (W, x_i, T) occurs in $\mathsf{eva}_{\mathsf{T}}(i)_0^A$, and (W, x_i, F) occurs in $\mathsf{eva}_{\mathsf{T}}(i)_1^A$. We also note that any $\mathsf{eva}_b(i)_r^A$ has all its reads of hb_i reading value r.

Definition 25 (n-Z). *For each fixed $n \geq 1$ the decision problem n-Z asks, given an n-counter machine A, if there exists $\sigma \in \mathcal{L}(A)$ such that σ is admitted.*

The problem n-Z is closely related to the halting problem for Minsky's so-called *program machines*, which also manipulate unbounded counters [101]. Program machines are Turing-complete when endowed with at least 2 counters. For n-Z we have a similar result, due to Alur et al. [8].

Lemma 20 (Alur et al.'s Theorem 1 [8]). *n-Z is undecidable for any $n \geq 3$.*

2.7.1 The Protocol $\mathcal{P}_\mathcal{C}$

In this section we show how, given any n-counter machine $\mathcal{C}$, to construct a protocol $\mathcal{P}_\mathcal{C}$ such that $\mathcal{C}$ has no admitted string if and only if $\mathcal{P}_\mathcal{C}$ is SC; our theorem then follows by Lemma 20. The construction borrows heavily from the proof of Alur et al. [8] that SC is undecidable when the protocols are *not* restricted to be prefix-closed.[23]

Key to the construction is the observation that $\sigma \in \Sigma_n^*$ is *not* admitted if and only if there is a way to shuffle σ into a string σ' such that the following two conditions hold.

Condition 1. *σ' is a member of the regular language*

$$\bigcup_{i=1}^{n}((\mathsf{I}_i \cup \mathsf{D}_i)^*\mathsf{Z}_i)^*(\mathsf{I}_i\mathsf{D}_i)^*(\mathsf{I}_i^+ \cup \mathsf{D}_i^+)\mathsf{Z}_i\Sigma_n^* \tag{2.16}$$

Condition 2. *For each $1 \leq i \leq n$, we have that $\sigma' \uparrow \{\mathsf{I}_i, \mathsf{Z}_i\} = \sigma \uparrow \{\mathsf{I}_i, \mathsf{Z}_i\}$ and $\sigma' \uparrow \{\mathsf{D}_i, \mathsf{Z}_i\} = \sigma \uparrow \{\mathsf{D}_i, \mathsf{Z}_i\}$. Equivalently, the shuffling cannot commute I_i with Z_i nor D_i with Z_i, but may commute any other pairs of symbols.*

Let $\mathcal{J}$ be a finite automaton such that $\mathcal{L}(\mathcal{J})$ is (2.16). It is convenient if we add an end of string marker symbol \$ to all strings in both $\mathcal{L}(\mathcal{C})$ and $\mathcal{L}(\mathcal{J})$. This can be done by simply adding

[23] Other than the cosmetic, there are two primary differences between our construction and that of Alur et al. First, nontrivial modifications have been made so that the reduction works when protocols are defined to be prefix-closed. The original proof need only concern itself with defining a map ϕ such that $\phi(\sigma)$ has properties corresponding (roughly) to our Conditions 1 and 2 whenever $\sigma \in \mathcal{L}(\mathcal{C})$. Here we must be careful to ensure that our ϕ has the further attribute that all prefixes of $\phi(\sigma)$ are SC. The most salient modifications in support of this end are the use of the symbol placeholder γ', the prefix θ_0, and the use of a second "heartbeat" address hb'. Second, Alur et al. give a very general definition of SC that encompasses concurrent objects other than shared memory addresses; let us call this GSC. Alur et al.'s reduction from n-Z to SC takes a detour through GSC, i.e they reduce n-Z to GSC, and then GSC is reduced to SC. To avoid introducing GSC in this thesis, our reduction is directly from n-Z to SC.

a fresh final state, and creating a transition on \$ from each of the "old" final states to the new final state. Hence, we may assume that both $\mathcal{C}$ and $\mathcal{J}$ are finite automata over the alphabet $\Sigma_n^{\$} = \Sigma_n \cup \{\$\}$ and each have unique final states, and we write $\mathcal{C} = (S_{\mathcal{C}}, \Sigma_n^{\$}, \delta_{\mathcal{C}}, s_{\mathcal{C}}, \{f_{\mathcal{C}}\})$, and $\mathcal{J} = (S_{\mathcal{J}}, \Sigma_n^{\$}, \delta_{\mathcal{J}}, s_{\mathcal{J}}, \{f_{\mathcal{J}}\})$.

The protocol $\mathcal{P}_{\mathcal{C}}$ will be defined so that its language is the prefix-closure of $\{\phi(\sigma) \mid \sigma \in \mathcal{L}(\mathcal{C})\}$, where $\phi : \Sigma_n^{\$ *} \to \mathcal{M}^*$ has the property that for any $\sigma \in \mathcal{L}(\mathcal{C})$,

- $\phi(\sigma)$ is SC if and only if σ is *not* admitted, and
- any proper prefix of $\phi(\sigma)$ is always SC.

Intuitively, $\phi(\sigma)$ maps each symbol α in σ to a string of reads and writes that encode both α and *all* transitions that $\mathcal{J}$ has on α. This is done in such a way so that if σ contains no \$, then $\phi(\sigma)$ is trivially SC. However, if $\sigma = \sigma_0\$$ (where $\sigma_0 \in \Sigma_n^*$), then $\phi(\sigma)$ will have a serial-reordering π if and only if $\phi(\sigma)^\pi$ represents an accepting run of $\mathcal{J}$ on a shuffling σ' of σ (i.e. σ' satisfies Condition 1) such that σ' also obeys Condition 2.

We will now define our mapping $\phi : \Sigma_n^{\$ *} \to \mathcal{M}^*$, in terms of several other functions. All of these definitions call upon the atomicity construct, previously described in Def. 8. The first, ρ, encodes a transition (s, α, s') of $\mathcal{J}$ as follows.

$$\rho((s, \alpha, s')) = \big\langle (R, \mathsf{st}, s)(R, \mathsf{sym}, \alpha)(W, \mathsf{st}, s')(W, \mathsf{sym}, \epsilon) \big\rangle_{(s,\alpha,0)}$$

The two read operations guarantee that if (s, α, s') is used in the accepting run of $\mathcal{J}$ on σ', then the transition must only occur when $\mathcal{J}$ is in state s (indicated by s residing in address st) and the next symbol is α (indicated by α residing in address sym). The two writes ensure that, after the transition, the state is s' (by storing s' to address st) and the symbol α is consumed (by storing the null symbol ϵ at sym).

Unused transitions require place-holders to satisfy their reads, and to ensure that they occur in the 1-phase of hb. The placeholder of a transition is built using the function ρ', where

$$\rho'((s, \alpha, s')) = \langle (R, \mathsf{hb}, 1)(W, \mathsf{st}, s)(W, \mathsf{sym}, \alpha) \rangle_{(s,\alpha,1)}$$

The encoding of a symbol $\alpha \in \Sigma_n$ is accomplished via the function γ, defined here.

$$\gamma(\alpha) = \begin{cases} \langle (R, \mathsf{sym}, \epsilon)(R, \mathsf{hb}, 0)(W, \mathsf{sym}, \mathsf{I}_i) \rangle_{2i} & \text{if } \alpha = \mathsf{I}_i \\ \langle (R, \mathsf{sym}, \epsilon)(R, \mathsf{hb}, 0)(W, \mathsf{sym}, \mathsf{D}_i) \rangle_{2i+1} & \text{if } \alpha = \mathsf{D}_i \\ \langle (R, \mathsf{sym}, \epsilon)(R, \mathsf{hb}, 0)(W, \mathsf{sym}, \mathsf{Z}'_i) \rangle_{2i} \langle (R, \mathsf{sym}, \mathsf{Z}'_i)(W, \mathsf{sym}, \mathsf{Z}_i) \rangle_{2i+1} & \text{if } \alpha = \mathsf{Z}_i \\ \langle (R, \mathsf{sym}, \epsilon)(R, \mathsf{hb}, 0)(W, \mathsf{sym}, \$) \rangle_{1} & \text{if } \alpha = \$ \end{cases}$$

For any α, $\gamma(\alpha)$ involves a reading of ϵ from sym; this ensures that in any reordering, $\gamma(\alpha)$ follows a transition encoding (rather than another symbol encoding). The read of hb ensures that all symbol encodings occur in the 0-phase of hb. Encoding Z_i involves actions of both the processors $2i$ and $2i+1$ so that encodings of neither I_i nor D_i can commute with Z_i in a reordering. The writing and reading of the new symbol Z'_i forces the actions of the two different processors to be paired up in any reordering.

Unlike the proof in [8], we will require place-holders for symbols. The effect of these place-holders is to ensure that for any proper prefix σ_0 of a string $\sigma \in \mathcal{L}(\mathcal{C})$, $\phi(\sigma_0)$ is trivially SC. The symbol place-holders must occur in the 1-phase of hb', and are used to satisfy the reads of the symbols encodings. The same symbol placeholder is used for all symbols, and is

$$\gamma' = \langle (W, \mathsf{sym}, \epsilon) \rangle_{\mu}$$

Given a $\alpha \in \Sigma_n$, we define

$$\psi_\rho(\alpha) = \rho'(t_1)\rho(t_1) \dots \rho'(t_k)\rho(t_k)$$

where $\{t_1, \dots, t_k\}$ are all the transitions of $\mathcal{J}$ on α; Each symbol α in σ is expanded into a place-holder for α, the encoding of α, and $\psi_\rho(\alpha)$ as follows:

$$\psi(\alpha) = \gamma' \gamma(\alpha) \psi_\rho(\alpha)$$

Finally we are equipped to define ϕ

$$\phi(\sigma) = \theta_0 \psi(\sigma(1)) \psi(\sigma(2)) \dots \psi(\sigma(|\sigma|))(R, \nu, \mathsf{hb}', 0)$$

where the prefix θ_0 is

$$\theta_0 = \big\langle (W, \mathsf{hb}, 0)(W, \mathsf{hb}', 0)(W, \mathsf{st}, s_{\mathcal{J}})(W, \mathsf{sym}, \epsilon) \big\rangle_{\nu} \langle (W, \mathsf{hb}, 1) \rangle_{\nu} \big\langle (W, \mathsf{hb}', 1) \big\rangle_{\mu}$$

To define $\mathcal{P}_{\mathcal{C}}$, we allow our protocols to have strings over $\mathcal{M}$ labeling their transitions. Clearly such a protocol can be easily transformed into one that has only single memory actions labeling its transitions by adding more states. $\mathcal{P}_{\mathcal{C}}$ is defined by $(S_{\mathcal{C}} \cup \{r, f\}, \mathcal{M}^*, \rightarrow, r)$, where r and f are new states not in $S_{\mathcal{C}}$, and

$$\rightarrow \;=\; \left\{(s, \psi(\alpha), s') \mid (s, \alpha, s') \in \delta_{\mathcal{C}}\right\} \cup \left\{(r, \theta_0, s_{\mathcal{C}}), (f_{\mathcal{C}}, (R, \nu, \mathsf{hb}', 0), f)\right\}$$

Lemma 21. *$traces(\mathcal{P}_{\mathcal{C}}) \;=\; \{\tau \mid \exists \sigma \in \mathcal{L}(\mathcal{C}) : \tau$ is a prefix of $\phi(\sigma)\}$*

Proof. Easy. □

2.7.2 The Proof

Here we state and prove our main result of this section.

Theorem 11. *Given a protocol $\mathcal{P}$, the problem of checking if $\mathcal{P}$ is SC is undecidable.*

Proof. By a reduction from n-Z. Given an n-counter machine $\mathcal{C}$ where $n \geq 3$, we construct a protocol $\mathcal{P}_{\mathcal{C}}$ as described in Sect. 2.7.1. By Lemma 22, $\mathcal{P}_{\mathcal{C}}$ is SC if and only if $\mathcal{L}(\mathcal{C})$ contains no admitted strings. The theorem follows from the fact that n-Z is undecidable (Lemma 20). □

Lemma 22. *Let $\mathcal{C}$ be an n-counter machine for some n. Then $\mathcal{P}_{\mathcal{C}}$ is SC if and only if $\mathcal{L}(\mathcal{C})$ contains no admitted strings.*

Proof. We first show that a certain set of traces of $\mathcal{P}_{\mathcal{C}}$ (the "unconsummated" traces) are always SC, regardless of whether or not $\mathcal{L}(\mathcal{C})$ contains admitted strings, hence we need only consider traces outside of this set. We say that $\tau \in traces(\mathcal{P}_{\mathcal{C}})$ is *consummated* if $\tau = \phi(\sigma)$ for some $\sigma \in \mathcal{L}(\mathcal{C})$, otherwise we will say that τ is *unconsummated*. By Lemma 21, any unconsummated $\tau \in traces(\mathcal{P}_{\mathcal{C}})$ is the prefix of some consummated trace of $\mathcal{P}_{\mathcal{C}}$. Clearly, $\tau \in traces(\mathcal{P}_{\mathcal{C}})$ is consummated if and only if the final event of τ is $(R, \nu, \mathsf{hb}', 0)$. Let τ be a *maximal* unconsummated trace of $\mathcal{P}_{\mathcal{C}}$, i.e $\phi(\sigma) = \tau(R, \nu, \mathsf{hb}', 0)$ for some $\sigma \in \mathcal{L}(\mathcal{C})$. Then the following specifies a serial

reordering of τ

$$\begin{aligned}
&\big\langle (W,\mathsf{hb},0)(W,\mathsf{hb}',0)(W,\mathsf{st},s_{\mathcal{J}})(W,\mathsf{sym},\epsilon)\big\rangle_\nu \\
&\big\langle (W,\mathsf{hb}',1)\big\rangle_\mu \\
&\textstyle\prod_{i=1}^{|\sigma|} \gamma'\gamma(\sigma(i)) \\
&\langle (W,\mathsf{hb},1)\rangle_\nu \\
&\textstyle\prod_{i=1}^{|\sigma|} \psi_\rho(\sigma(i))
\end{aligned} \tag{2.17}$$

The reader may confirm that (2.17) is not only a serial-reordering of τ, but is in fact a DSC-reordering. Hence, restricting (2.17) to any prefix τ' of τ gives a DSC-reordering for τ'. Since any unconsummated trace is a prefix of some consummated trace, we conclude that all unconsummated traces of $\mathcal{P}_\mathcal{C}$ are SC (in fact they are DSC).

Thus we need only concern ourselves with consummated strings. We will argue that for any consummated $\tau \in \mathit{traces}(\mathcal{P}_\mathcal{C})$, τ is SC if and only if σ is not admitted, where $\tau = \phi(\sigma)$.

($\Rightarrow$) Let us suppose that τ is SC, and consider some serial reordering τ^π. Because of the existence of the event $(R,\nu,\mathsf{hb}',0)$, the 0-phase of hb and the 1-phase of hb' are disjoint, and the latter appears later. All occurrences of $\gamma(\alpha)$ must occur in the 0-phase of hb, and all occurrences of γ' must occur in the 1-phase of hb'. Hence all symbol place-holders occur after all symbol encodings. Furthermore, for any $1 \le i \le n$, each occurrence of $\gamma(\mathsf{Z}_i)$ must occur atomically in τ^π. To see this, let $\gamma_1(\mathsf{Z}_i)$ and $\gamma_2(\mathsf{Z}_i)$ respectively be the first and second atomicity construct of $\gamma(\mathsf{Z}_i)$. The only writes of Z_i' to sym in τ^π happen in occurrences of $\gamma_1(\mathsf{Z}_i)$. The read and write of sym in $\gamma_2(\mathsf{Z}_i)$ thus ensure that all occurrences of $\gamma_1(\mathsf{Z}_i)$ and $\gamma_2(\mathsf{Z}_i)$ strictly alternate in τ^π. Since each $\gamma_1(\mathsf{Z}_i)$ must occur in the 0-phase of hb, the only possible atomicity construct that could occur between an occurrence of $\gamma_1(\mathsf{Z}_i)$ the the following $\gamma_2(\mathsf{Z}_i)$ is a transition encoding ρ or some other symbol encoding γ. However, either of these would overwrite the Z_i' stored in sym, which would prohibit the read of sym in $\gamma_2(\mathsf{Z}_i)$ from receiving the value written by $\gamma_1(\mathsf{Z}_i)$.

It follows that

$$\begin{aligned}\tau^{\pi} = \quad & \big\langle (W,\mathsf{hb},0)(W,\mathsf{hb}',0)(W,\mathsf{st},s_{\mathcal{J}})(W,\mathsf{sym},\epsilon)\big\rangle_{\nu} \\ & \textstyle\prod_{j=1}^{|\sigma|}\gamma(\sigma(i_j))\rho((s_{j-1},\sigma(i_j),s_j)) \\ & \langle (W,\mathsf{hb},1)\rangle_{\nu} \\ & (R,\nu,\mathsf{hb}',0) \\ & \big\langle (W,\mathsf{hb}',1)\big\rangle_{\mu} \\ & \eta\end{aligned} \tag{2.18}$$

Here $\sigma' = \sigma(i_1)\sigma(i_2)\ldots\sigma(i_{|\sigma|})$ is a reordering of σ that satisfies both Condition 1 and Condition 2, and η is a string involving: all symbol place-holders, all transition place-holders, and all transition encodings *except* the $|\sigma|$ transitions used prior to $(W,\nu,\mathsf{hb},1)$. To see that Condition 1 holds, note that

$$s_0, \sigma(i_1), s_1, \sigma(i_2), s_2, \ldots, \sigma(i_{|\sigma|}), s_{|\sigma|}$$

is a run of $\mathcal{J}$ on σ'. That $s_0 = s_{\mathcal{J}}$ follows from the fact that the initial write to st stores $s_{\mathcal{J}}$. That $s_{|\sigma|} = f_{\mathcal{J}}$ follows from the fact that σ has exactly one occurrence of \$, and all transitions of $\mathcal{J}$ on \$ have $f_{\mathcal{J}}$ as the final component. Furthermore, there are no transitions with $f_{\mathcal{J}}$ in the first component. Thus the reordering only works if $\sigma(i_{|\sigma|}) = \$$ and $s_{|\sigma|} = f_{\mathcal{J}}$. Condition 2 is guaranteed by the processor names used in defining γ, and the fact that π preserves per-processor order. Therefore σ is not admitted.

($\Leftarrow$) Conversely if σ is not admitted, then Conditions 1 and 2 holds of σ, and it follows that we can construct a serial-reordering of τ of the form (2.18). $\square$

Chapter 3

Address and Value Parameterization

3.1 Introduction

In Chapter 2, we defined a restriction of SC called DSC, and considered the computational complexities of several related verification problems. A protocol was defined to be a prefix-closed finite state automaton, having some transitions labelled with memory events (page 13). Furthermore, a protocol only includes memory events pertaining to a fixed, finite number of processors, addresses, and data values. A real protocol description, however, is typically *parameterized* by these (and typically other) quantities, meaning that the description defines a protocol for any arbitrary numbers of processors, addresses, and data values.

This chapter presents a method for proving sequential consistency of an infinite family of protocols parameterized in two dimensions: the number of addresses, and the number of data values. We consider the number of processors to be a fixed constant. The approach is abstraction-based, and we show that an appropriate abstraction can be extracted automatically from the protocol family description. Using the assumption of data independence, verification over the data value parameter is relatively straightforward; our main contribution is a means to handle parameterized addresses.

In theory, handling a parameterized number of processors is most interesting, because shared memory protocols are intended to facilitate complex interactions among processors (later, in Chapter 4, we develop an approach that can be used for processor-parameterized model checking in some circumstances). In practice, however, handling parameterized numbers of addresses and data values is a higher priority, because real shared-memory multiprocessors have few processors and many addresses and data values. Indeed, many multiprocessors are chip multiprocessors in which 2 or 4

$$\begin{array}{c|llll} p_1 & (W,a_2,1)(W,a_1,2) & & (R,a_2,1)(R,a_2,2) & \\ p_2 & & (W,a_1,1)(W,a_2,2) & & (R,a_1,1)(R,a_1,2) \end{array}$$

Figure 3.1: Example trace τ that is not SC. The values seen by the two reads on one processor imply that the other processor's second write appears to have occurred between the two reads; no reordering can satisfy this property for both processors simultaneously. However, both of $\tau \uparrow (*, *, a_1, *)$ and $\tau \uparrow (*, *, a_2, *)$ satisfy the stronger property of SSC (see Def. 27). This demonstrates nothing interesting (for our purposes!) can be inferred about a trace from the fact that its projection onto each address is always SSC.

CPUs are collocated on the same die. In contrast, the smallest and most common configurations have 2^{32} data values and hundreds of millions of physical addresses — far beyond the reach of the direct application of model checking.

Our method leverages a few common, practical assumptions about memory system protocols. Three of these — data independence, processor symmetry, and address symmetry — are standard and easily enforced syntactically. We impose some additional syntactic constraints to simplify the automatic generation of a finite-state abstract protocol from the parameterized protocol description; these are described in Sect. 3.4. We prove that correctness of this abstract protocol implies that of the infinite family of protocols. Checking correctness of the abstract protocol can be dispatched to any of the existing techniques.

The bulk of this chapter represents work published by the author et al. [19] with several nontrivial notational and terminological changes along with a clean-up of the exposition. Also, Sect. 3.7 provides some relevant insights that did not appear in that publication.

3.2 Preliminaries

Def. 5 defined the value-parameterized protocol family; here we make a similar definition to capture protocols parameterized both by the number of addresses and number of data values.

Definition 26 (address and value-parameterized protocol family). *An* address and value parameterized protocol family *(AVPPF) is a function $\mathcal{F}$ such that for any $m, v \geq 1$ we have that $\mathcal{F}(m, v)$ is a protocol involving n processors, m addresses, and v data values, where $n \geq 1$ is constant.*

Our method verifies a stronger form of SC in which writes to the same address cannot be re-

ordered.[1] To our knowledge, all implemented SC protocols implement this stronger form. (The canonical example of a protocol that is SC, but violates this assumption is *Lazy Caching* [6], which was described in Sect. 2.3.3) With these restrictions, our method is, in principle, fully automatic.

Definition 27 (simple serial reordering, simple SC). *A serial reordering π of a trace τ is said to be a* simple serial reordering *(of τ) if for each address a appearing in τ we have*

$$\tau^{\pi} \uparrow (W, *, a, *) \;=\; \tau \uparrow (W, *, a, *)$$

A trace is said to be simple sequentially consistent *(SSC) if it has a simple serial reordering.*

Intuitively, SC says that there must exist a reordering that is serial and preserves the per-processor order. SSC adds the requirement that the ordering of writes to each address is also preserved in the reordering. As is expected, a protocol is deemed SSC if all its traces are SSC. Similarly an AVPPF is said to be SSC if all protocols in its image are SSC. The notion of SSC has appeared previously as the *real-time store reordering property* of Condon and Hu [36] and as the *simple witness* of Qadeer [113]; both of these works employ SSC as a simplifying assumption in their respective verification methods (as do we).

We note that, like SC, SSC cannot be checked on a "per-address" basis. In fact, there exist non-SC traces involving only two addresses such that projecting onto either address yields an SSC trace. Fig. 3.1 provides such a trace. Hence verifying SSC for an arbitrary number of addresses is a formidable challenge.

We now define three common protocol assumptions we make on AVPPFs: address symmetry, processor symmetry, and data independence. For defining the symmetries, let us fix a finite set of processors P and a finite set of addresses A. For any permutation λ on P, define λ^{proc} to be the function on $\mathcal{M}(P, A, \mathbb{N})$ specified by $\lambda^{proc}((o, p, a, d)) = (o, \lambda(p), a, d)$. Similarly, if λ is a permutation on A, define $\lambda^{addr}((o, p, a, v)) = (o, p, \lambda(a), v)$. We extend λ^{proc} and λ^{addr} to have domain and range $\mathcal{M}^*$ in the obvious way.

Definition 28 (address and processor symmetry). *Let $\mathcal{P}$ be a protocol involving the set of processors P and the set of addresses A. Then $\mathcal{P}$ is* address symmetric *if for every permutation $\lambda : A \rightarrow A$ we have $\tau \in traces(\mathcal{P})$ implies $\lambda^{addr}(\tau) \in traces(\mathcal{P})$. Similarly, $\mathcal{P}$ is* processor symmetric *if for*

[1] In Chapter 6 we propose a method that might be able to lift this restriction.

every permutation $\lambda : P \to P$ *we have* $\tau \in traces(\mathcal{P})$ *implies* $\lambda^{proc}(\tau) \in traces(\mathcal{P})$. *An AVPPF is said to be address symmetric or processor symmetric if all protocols in its image have the respective property.*

Data independence was defined with respect to value-parameterized protocol families in Def. 7 on page 18. The definition extends naturally to the AVPPF $\mathcal{F}$. For each $m \geq 1$, let $\mathcal{F}_m$ be the value-parameterized protocol family defined by $\mathcal{F}_m(v) = \mathcal{F}(m, v)$ for all $v \geq 1$. Then we say that $\mathcal{F}$ is data independent if for each $m \geq 1$, $\mathcal{F}_m$ is data independent according to Def. 7.

3.3 Verification Approach

Our aspiration is to verify that an AVPPF $\mathcal{F}$ is SSC. This section presents Theorem 12, which states that SSC of $\mathcal{F}$ can be soundly reduced to SSC of a single protocol Q involving n processors, n addresses, and 3 data values, provided that an infinite number of projected trace containments hold between certain members of $\mathcal{F}$ and Q (see condition 2 of Theorem 12). Here and throughout the remainder of the chapter, n denotes the fixed number of processors of $\mathcal{F}$. Thus (roughly) we reduce correctness for an arbitrary number of addresses and data values to correctness for only n addresses and 3 data values. In Sect. 3.5 we will show that if $\mathcal{F}$ is expressed in a certain formalism, then we can effectively produce a Q for which these containments hold "by construction". Then, SSC of Q can be checked using known methods based on model-checking [18, 22, 36, 113].

The proof of Theorem 12 relies on machinery developed by Nalumasu [103] and later Qadeer [113] to show that, under the address symmetry, processor symmetry, and data independence assumptions, any trace τ that is not SSC can be "proven" not SSC by considering only its projection onto the addresses in some set A, where $|A|$ is bounded by the number of processors in τ. Since our family has a fixed number of processors n, this means that for a trace with an arbitrarily large number of addresses, we need only to consider its projection onto a small set of addresses. By exploiting the symmetry assumptions, a violating trace can always be found such that $A = \mathbb{N}_n$. Furthermore, using data independence, we can ensure that a violating trace exists with 3 data values. Hence, if we can construct a protocol Q such that $traces(Q)$ contains all projections of traces of our family involving $m > n$ addresses and 3 data values, then SSC of Q implies that of $\mathcal{F}(m, v)$ for all $m > n$ and all $v \geq 1$. This reduction from SSC of an AVPPF $\mathcal{F}$ to that of a

single finite state protocol Q is formally stated in Theorem 12. We now develop the definitions and lemmas that support the proof of this theorem.

Recall from Sect. 2.3.2 that an *unambiguous* trace is one in which no two writes to the same address write the same value. Given an unambiguous trace, we can construct a digraph that completely captures all ordering constraints any simple serial reordering of the trace must respect. The following definition is based on the notion of a trace *constraint graph* [36, 66, 113].

Definition 29 (simple constraint graph). *Given an unambiguous trace τ, the* simple constraint graph *of τ is the digraph $\mathsf{scg}(\tau) = (V, E)$ where $V = \mathbb{N}_{|\tau|}$, and $E \subseteq V \times V$ is the least relation such that:*

i) $x < y$ *and* $proc(\tau(x)) = proc(\tau(y))$ *imply* $(x, y) \in E$

ii) whenever $op(\tau(x)) = W$, $op(\tau(y)) = R$, $addr(\tau(x)) = addr(\tau(y))$, *and* $val(\tau(x)) = val(\tau(y))$, *we have* $(x, y) \in E$

iii) for any x and y such that there exists x' and y' such that $x' < y'$, $op(\tau(x')) = op(\tau(y')) = W$, $addr(\tau(x)) = addr(\tau(x')) = addr(\tau(y)) = addr(\tau(y'))$, and $val(\tau(x)) = val(\tau(x')) \neq val(\tau(y)) = val(\tau(y'))$, we have $(x, y) \in E$

We say an edge in a constraint graph is of *type t* if it is included as a result of item $t \in \{\text{i}, \text{ii}, \text{iii}\}$ from Def. 29.[2] Intuitively, type i edges enforce the constraint that serial reorderings must preserve the per-processor order. Type ii edges enforce the constraint that whenever a read inherits its value from a write, the read must be reordered after the write; note that because of unambiguity, there is a unique write that any read can inherit from. A type iii edge enforces the constraint that in a simple serial reordering, writes to the same address cannot be reordered, and furthermore, *any* two events with the same address but different data values must come in the same order as the two unique writes with the respective values.

Lemma 23. *An unambiguous trace τ is SSC if and only if $\mathsf{scg}(\tau)$ is acyclic.*

Proof. ($\Rightarrow$) If $\mathsf{scg}(\tau)$ has a cycle it follows that any reordering of τ will either not be serial, not respect per-processor order, or not respect the per-address write order of τ. ($\Leftarrow$) If $\mathsf{scg}(\tau)$ is acyclic

[2] Simple constraint graph edges don't necessarily have a unique type. For example, the edge between two writes by the same processor to the same address is both of type i and iii.

then it is an irreflexive partial order. Any total order on $\mathbb{N}_{|\tau|}$ that extends this partial order defines a simple serial reordering of τ. □

It turns out that if a simple constraint graph has a cycle then it has a certain type of cycle called a *k-nice cycle* [113]. A k-nice cycle is a cycle of length $2k$ with edges alternating between type i edges and edges of any type having endpoints with the same address. Also, each processor and each address appears in either exactly 0 or exactly 2 nodes in the cycle, and in the latter case the 2 nodes are consecutive.

Definition 30 (k-nice cycle [113]). *Let τ be an unambiguous trace, and let k be a positive integer. A k-*nice cycle *in $\mathsf{scg}(\tau)$ is a cycle $u_1, v_1, u_2, v_2, \ldots, u_k, v_k$ of length $2k$ such that*

1. *for each $x \in \mathbb{N}_k$ we have $proc(\tau(u_x)) = proc(\tau(v_x))$ and $addr(\tau(v_x)) = addr(\tau(u_{(x \mod k)+1}))$*
2. *for each $x, y \in \mathbb{N}_k$ we have $x \neq y$ implies $proc(\tau(u_x)) \neq proc(\tau(u_y))$*
3. *for each $x, y \in \mathbb{N}_k$ we have $x \neq y$ implies $addr(\tau(v_x)) \neq addr(\tau(v_y))$*

Lemma 24. *Let $\tau \in \mathcal{M}(\mathbb{N}_n, \mathbb{N}_m, \mathbb{N})^*$ be unambiguous. Then $\mathsf{scg}(\tau)$ has a cycle if and only if $\mathsf{scg}(\tau)$ has a k-nice cycle for some $k \in \{1, \ldots, \min(n, m)\}$.*

Proof. This lemma is a special case of Qadeer's Theorem 6.1 [113]. The proof uses the fact that any cycle can be reduced to a k-nice cycle by "cutting-out" extraneous segments of the cycle. □

Our proof of Theorem 12 below uses the fact that under the assumptions of processor and address symmetry, the existence of a cycle entails the existence of a k-nice cycle that is canonical in the following sense.

Definition 31 (canonical k-nice cycle). *A k-nice cycle $u_1, v_1, \ldots, u_k, v_k$ is said to be a* canonical *k-*nice cycle *if for each $x \in \mathbb{N}_k$ we have $proc(\tau(u_x)) = addr(\tau(u_x)) = x$.*

A canonical k-nice cycle only involves processors and addresses in the set $\mathbb{N}_k$. Essential to our proof of Theorem 12 is the fact that a canonical k-nice cycle will still exist if we project out (of the trace) events of processors and addresses that are not in $\mathbb{N}_k$; this is formalized by Lemma 25 below. Lemma 26, in some sense, does for data values what Lemma 25 does for addresses and processors. Lemma 26 asserts that given any unambiguous trace that has a cycle in its simple constraint graph,

one can rename the data values in the trace using only 3 values such that the resulting trace also violates SSC. Note that in general, the trace obtained through this renaming will be ambiguous, thus it does not have a well-defined simple constraint graph. Nevertheless, it can be proven that the renamed trace is not SSC.

Lemma 25. *Let τ be an unambiguous trace and $k \geq 1$ be such that* $\mathsf{scg}(\tau)$ *has a canonical k-nice cycle. Then for any $n \geq k$ we have that $\tau \uparrow (*, *, \mathbb{N}_n, *)$ is unambiguous and not SSC.*

Proof. Let $\tau' = \tau \uparrow (*, \mathbb{N}_n, \mathbb{N}_n, *)$. Clearly τ' is unambiguous, since removing events cannot create ambiguity. We show that $\mathsf{scg}(\tau')$ has a canonical k-nice cycle, thus the lemma follows by Lemma 23. Let $f : \mathbb{N}_{|\tau|} \to \mathbb{N}_{|\tau'|}$ be the partial function that takes the index of each occurrence of an event of $(*, \mathbb{N}_n, \mathbb{N}_n, *)$ in τ to its index in τ'; thus $f(i)$ is undefined iff $\tau(i) \notin (*, \mathbb{N}_n, \mathbb{N}_n, *)$. Let $c = (u_1, v_1, \ldots, u_k, v_k)$ be the canonical k-nice cycle of $\mathsf{scg}(\tau)$. Since $n \geq k$, f is defined for all vertices of c. We claim that $c' = (f(u_1), f(v_1), \ldots, f(u_k), f(v_k))$ is a canonical k-nice cycle in $\mathsf{scg}(\tau')$. Let E and E' be the set of edges of $\mathsf{scg}(\tau)$ and $\mathsf{scg}(\tau')$, respectively. By condition i of Def. 29 we have $(f(u_i), f(v_i)) \in E'$ for each $i \in \mathbb{N}_k$. Now let $e = (v_i, u_{(i \mod k)+1})$ and $e' = (f(v_i), f(u_{(i \mod k)+1}))$ for some $i \in \mathbb{N}_k$. e is included in E because of (at least) one of conditions i, ii, or iii of Def. 29. Since $addr(\tau(v_i)) = addr(\tau(u_{(i \mod k)+1}))$, whichever of these three conditions places e in E will also place e' in E'. Hence c' is a cycle in $\mathsf{scg}(\tau')$, and is clearly a canonical k-nice cycle. □

Lemma 26. *Let $\tau \in \mathcal{M}(\mathbb{N}_n, \mathbb{N}_m, \mathbb{N}_v)$ be an unambiguous trace and let $k \geq 1$ be such that* $\mathsf{scg}(\tau)$ *has a k-nice cycle. Then there exists a renaming function*[3] $\lambda : \mathbb{N}_m \times \mathbb{N}_v \to \mathbb{N}_3$ *such that $\lambda(\tau)$ is not SSC.*

Proof. The renaming function defined in the ($\Rightarrow$) direction of the proof of Qadeer's Theorem 8.1 [113] has this property[4]. □

Assuming data independence, any value-parameterized protocol family can be concluded to be SSC if either all its unambiguous traces are SSC, or if its protocol with 3 data values is SSC. Lemmas 27 and 28 phrase these two results in terms of AVPPFs.

[3]Renaming functions were defined back in Def. 6.

Lemma 27. *Let $\mathcal{F}$ be a data independent AVPPF, and let m be a positive integer. If every unambiguous trace of $\mathcal{F}(m, v)$ is SSC for all $v \geq 1$, then $\mathcal{F}(m, v)$ is SSC for all $v \geq 1$.*

Proof. This lemma is essentially Qadeer's Theorem 4.1 [113] with "sequentially consistent" replaced with "SSC" and the proof is similar. Suppose for all $v \geq 1$, every unambiguous trace of $\mathcal{F}(m, v)$ is SSC. Let τ be any trace of $\mathcal{F}(m, v)$ for some fixed v. By data independence, there exists a $v' \geq 1$, an unambiguous trace $\tau' \in traces(\mathcal{F}(m, v'))$, and a renaming function $\lambda : \mathbb{N}_m \times \mathbb{N}_{v'} \to \mathbb{N}_v$ such that $\tau = \lambda(\tau')$. Since τ' is SCC, it has a simple serial reordering π. Clearly π is also a simple serial reordering of τ. $\square$

Lemma 28. *Let $\mathcal{F}$ be a data independent AVPPF, and let m be a positive integer. If $\mathcal{F}(m, 3)$ is SSC then $\mathcal{F}(m, v)$ is SSC for all $v \geq 1$.*

Proof. Suppose there exists $v \geq 1$ such that $\mathcal{F}(m, v)$ is not SSC. By Lemma 27, there exists an unambiguous trace τ of $\mathcal{F}(m, v')$ for some v' such that τ is not SSC. From Lemma 24, $\mathsf{scg}(\tau)$ has a k-nice cycle for some $1 \leq k \leq \min(n, m)$. Using a construction similar to that of the proof of Qadeer's Theorem 5 [113], we can define a renaming function that maps τ to a (possibly ambiguous) trace $\tau' \in \mathcal{M}(\mathbb{N}_n, \mathbb{N}_m, \mathbb{N}_3)$ that is not SSC.[4] By data independence, τ' is a trace of $\mathcal{F}(m, 3)$, hence $\mathcal{F}(m, 3)$ is not SSC. $\square$

Theorem 12. *Let $\mathcal{F}$ be an AVPPF that is processor symmetric, address symmetric, and data independent. If there exists a protocol Q such that*

1. *Q is SSC, and*
2. *For all $m > n$ we have that $traces(\mathcal{F}(m, 3)) \uparrow \mathcal{M}(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3) \subseteq traces(Q)$*

then $\mathcal{F}(m, v)$ is SSC for all $v \geq 1$ and $m > n$.

Proof. We argue that if conditions 1 and 2 hold, then for all $m > n$ we have $\mathcal{F}(m, 3)$ is SSC, which implies the consequence of the theorem by Lemma 28. Fig. 3.2 depicts the relationships between

[4] Roughly, this renaming function does the following. For any address not involved in the cycle, the renaming function always maps to data value 1. Let $u_1, v_1, u_2, v_2, \ldots, u_k, v_k, u_{k+1} = u_1$ be the k-nice cycle. It follows from our definitions that for each $x \in \mathbb{N}_k$, there exists a unique $w_x \in \mathbb{N}_{|\tau|}$ such that $op(\tau(w_x)) = W$ and $val(\tau(w_x)) = val(\tau(u_x))$. For $x \in \mathbb{N}_k$ the renaming function maps the values written to address $addr(\tau(u_x))$ to the set $\{1, 2, 3\}$ as follows: values of writes that are prior (in τ) to $\tau(w_x)$ are mapped to 1, the value written by $\tau(w_x)$ maps to 2, and values of writes occurring after $\tau(w_x)$ map to 3.

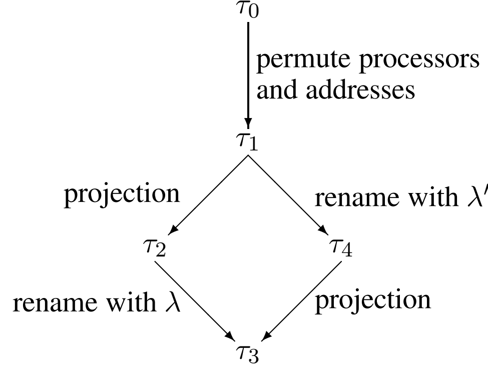


Figure 3.2: Relationship between the traces mentioned in the proof of Theorem 12. τ_0 is an unambiguous trace of $\mathcal{F}(m, v)$ that is not SSC. τ_1 is τ_0 with processors and addresses permuted so that $\mathsf{scg}(\tau_1)$ has a canonical k-nice cycle. In the proof we reduce τ_1 to a trace τ_3 such that τ_3 is not SSC (proven by way of τ_2), and τ_3 is a trace of Q (proven by way of τ_4). This implies that Q is not SSC.

the five traces $\tau_0, \ldots, \tau_4$ involved in this proof. Suppose $\mathcal{F}(m, 3)$ is not SSC for some $m > n$. Then by Lemma 27, there exists $v \geq 1$ and an unambiguous trace $\tau_0 \in \mathit{traces}(\mathcal{F}(m, v))$ such that τ_0 is not SSC. By Lemmas 23 and 24, there exists a k-nice cycle in $\mathsf{scg}(\tau_0)$, for some k such that $1 \leq k \leq n = \min(n, m)$. Since $\mathcal{F}$ is both location symmetric and processor symmetric, by permuting the processors and addresses in τ_0 we can obtain an unambiguous trace τ_1 of $\mathcal{F}(m, v)$ such that $\mathsf{scg}(\tau_1)$ contains a canonical k-nice cycle. By Lemma 25, $\tau_2 = \tau_1 \uparrow (*, \mathbb{N}_n, \mathbb{N}_n, *)$ is unambiguous and not SSC. By Lemma 26, there exists a renaming function $\lambda : \mathbb{N}_n \times \mathbb{N}_v \to \mathbb{N}_3$. such that $\tau_3 = \lambda(\tau_2)$ is not SSC. Let $\lambda' : \mathbb{N}_m \times \mathbb{N}_v \to \mathbb{N}_3$ be any renaming function that agrees with λ over the domain $\mathbb{N}_n \times \mathbb{N}_v$. Then, because of data independence, $\tau_4 = \lambda'(\tau_1)$ is a trace of $\mathcal{F}(m, 3)$. Note that $\tau_3 = \tau_4 \uparrow (*, \mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3)$. Now if condition 2 holds, we have $\tau_3 \in \mathit{traces}(Q)$, and thus Q is not SSC and condition 1 fails. $\square$

We note that Theorem 12 does not allow us to conclude that $\mathcal{F}(m, v)$ is SSC when $m \leq n$. This is not a serious deficiency, since we may verify correctness of these members via a finite number of model checks. Practically, these are the uninteresting cases, since multiprocessors always have many more addresses than processors.

Theorem 12 is extremely useful, provided we have available to us a protocol Q satisfying the two conditions of the theorem's antecedent. The question remains, how do we concoct Q? One obvious possibility is to try $Q = \mathcal{F}(n, 3)$. This Q satisfies the first condition, otherwise $\mathcal{F}$ is buggy.

The second condition does not necessarily hold, however, it *does* hold of $\mathcal{F}(n, 3)$ assuming that $\mathcal{F}$ is *location monotonic* (LM). LM means that for all $m, v \geq 1$ and for all j such that $1 \leq j \leq m$ we have

$$traces(\mathcal{F}(m, v)) \uparrow \mathcal{M}(\mathbb{N}_n, \mathbb{N}_j, \mathbb{N}_v) \subseteq traces(\mathcal{F}(j, v))$$

LM seems to be a property one might expect of real families. However, in contrast to our three assumptions of Sect. 3.2, it is not clear how one might either *prove* LM or *enforce* LM through syntactic restrictions. LM is used as an unproved assumption by Henzinger et al. [74], i.e. soundness of their verifications are contingent on LM holding and no guarantees of LM are made. Nalumasu [103] also employs LM,[5] and provides a formalism for describing protocols that are guaranteed to have this property, however the expressiveness of this framework is unclear.

Instead of using $Q = \mathcal{F}(n, 3)$ along with LM, we advocate a different approach: Q is constructed as an abstraction of $\mathcal{F}(m, 3)$ for all $m > n$, such that the second condition of Theorem 12 is guaranteed to hold. The automatic construction of our Q is possible because of the formalism we use to describe $\mathcal{F}$ which is presented in Sect. 3.4. The first condition of the theorem is then checked algorithmically using known methods based on model-checking [18, 22, 36, 113]. If this check is successful, the conclusion of Theorem 12 follows. Otherwise, the approach has failed, and we can draw no conclusions; in other words the approach is *sound* but *incomplete*. Hence, to argue for the applicability of this approach, one must argue that real protocol families have the following properties:

Property 1. *They are address symmetric, processor symmetric, and data independent*

Property 2. *They are expressible in the formalism of Sect. 3.4*

Property 3. *They will not yield false negatives; i.e. if the family is indeed SSC then the approach succeeds*

It is widely accepted that real protocols satisfy Property 1 [113], especially at the high level where we model.[6] With regard to Property 2, we note that our formalism of Sect. 3.4 is quite general, encompassing all real protocols that we have encountered. For instance, many published

[5] Nalumasu calls LM *projectability*

[6] Even protocols that utilize some total order on processor names (a feature that seemingly breaks processor symmetry), can, in principle, be made processor symmetric by having the total order selected nondeterministically in the initial state.

protocols [6,13,16,69,84,93] are expressible in our formalism. In support of Property 3, we present successful experiments on two challenging protocols in Sect. 3.6. Furthermore, in Sect. 3.7.2 we describe a simple protocol family that *does not* have Property 3, and propose a work-around for such families requiring modest user assistance.

3.4 A Protocol Description Formalism

We present our AVPPF description formalism as a restricted first-order logic inspired by the *bounded-data parameterized systems* of Pnueli et al. [109]. It is not our intent that protocols are to be written in this logic, rather, that one describes the protocol in a typical formal modeling language (eg. Murφ [46], SMV [94], TLA+ [89], etc.) and our logic manifests as a set of syntactic restrictions on the language of choice. The formalism is tuned to provide enough expressiveness for the real protocol descriptions we have encountered, while still allowing the efficient and automatic generation of a candidate abstraction Q.

Theorem 12 requires the AVPPF to be processor symmetric, address symmetric, and data independent. All three of these properties can be enforced straightforwardly at the syntactic level. For instance, data independence can easily be enforced by restricting the operations performed on data values as observed by Wolper [124] and others [103, 113], while symmetric types such as Murφ scalarsets [79] can be used to ensure processor and address symmetry. Our challenge in defining our protocol description logic, then, is not in enforcing these properties, but in allowing for the automatic construction of Q such that the trace containments of condition 2 of Theorem 12 hold.

This serves as our justification for several simplifications we make in presenting our formalism. We *do not* provide special types for processor names and data values; we assume that this is done externally and in a manner that yields processor symmetry and data independence (address symmetry, however, is intrinsic to our syntax). In fact, other than the type of addresses and arrays indexed by addresses, we only support variables of boolean type. Hence processor names, data values, and other enumerated types must be encoded as booleans. Furthermore, since our proposed Q will abstract $\mathcal{F}(m, 3)$ for all $m > n$, we treat the number of data values as fixed at 3. Again, the understanding is that in the real description, data values will be of a parameterized type that is manipulated in a data independent manner. Therefore we are really only handling parameterization by the number of addresses, i.e. given any $m \geq 1$, our logic characterizes the protocol $\mathcal{F}(m, 3)$

involving a constant n processors, m addresses, and a constant 3 data values.

3.4.1 Syntax

We assume three disjoint sets of variables $X = \{x_1, \ldots, x_{|X|}\}$, $Y = \{y_1, \ldots, y_{|Y|}\}$, and $Z = \{z_1, \ldots, z_{|Z|}\}$; and denote $X \cup Y \cup Z$ by Vars. Priming any of these sets has the effect of priming all constituent variables; semantically, primed variables will represent the variable in the next state. The variables of X, Y, and Z are respectively of type Boolean, address, and array of Boolean indexed by address. We call the variables of Y *address ranged variables* (ARVs). We employ two auxiliary ARVs a and a_1 that will (effectively) be quantified over, and use ARVars to denote the set $Y \cup Y' \cup \{a, a_1\}$. In shared memory protocols, typical (though by no means exhaustive) usages of the variables in X, Y, and Z, are respectively: fields corresponding to processor IDs or data values or message types in messages, fields storing addresses in messages, and the permission bits and data value associated with each address in a local cache or a directory. Note that we include a special "null address" value nul in the range of variables in $Y \cup Y'$. This is convenient in uniquely defining an initial state, and also in directly handling languages (such as Murφ [46]) that support undefined values.

At the heart of our logic is the quantifier-free action, which is roughly analogous to a Murφ rule or a TLA+ action.

Definition 32 (quantifier-free action). *A* quantifier-free action *(QFA) is a formula with the syntax* Φ*, defined by:*

$$Atom \;::=\; x \mid z[a] \mid z[a_1] \mid \alpha{=}\beta \mid y{=}\mathsf{nul}$$
$$\Phi \;::=\; Atom \mid \neg\Phi \mid (\Phi \vee \Phi) \mid (\Phi \wedge \Phi)$$

where $x \in X \cup X'$*,* $y \in Y \cup Y'$*,* $z \in Z \cup Z'$*, and* $\alpha, \beta \in \mathsf{ARVars}$*. Also, terms of the syntax* $Atom$ *are called* atoms*, and atoms of the form* $\alpha{=}\beta$ *are called* ARV equality tests*.*

The transition relation of our family is expressed by a finite set of QFAs, each associated with a unique element from a finite set of *action labels* L such that $\{R, W\} \times \mathbb{N}_n \times \mathbb{N}_3 \subseteq L$. Formally, the transition relation is expressed by

$$\{\phi_\ell \mid \ell \in L\}$$

where, for each $\ell \in L$, ϕ_ℓ is a QFA. Aside from the transition relation, we must describe also the initial state of our AVPPF. This is done by a function $init : X \cup Z \to \mathbb{B}$, which takes each variable

in $X \cup Z$ to an initial boolean value. For $x \in X$, $init(x)$ specifies the initial value of x, while for $z \in Z$, $init(z)$ is the initial value of *every entry* in the array z. The variables of Y are not interpreted by $init$ because their initial values are required to be nul.

As we will see in Sect. 3.4.2, a transition (i.e. state/next-state pair) can be made only if it satisfies some ϕ_ℓ, with a existentially quantified and a_1 universally quantified. Thus a can be viewed as a distinguished address pertaining to the transition, while all other addresses a_1 are treated in a uniform manner. This restriction accords exactly with what we have observed in real protocol descriptions: typically, this uniform treatment of the "other" addresses a_1 asserts that state associated with a_1 (i.e. any array entry that *doesn't* correspond to a) remains fixed. However, several transitions in the WIS protocol we experiment with in Sect. 3.6 have a more involved effect on the other addresses, i.e. a certain state change for one address forces all other addresses to abandon an optimization mode in the relevant cache. A limitation of our formalism immediately evident from Def. 32 is that the auxiliary ARVs a and a_1 are the only ARVs that can be used to index into arrays. However, typical instances of indexing into some $z \in Z$ using an ARV $y \in Y$ can be performed by, for example, $y{=}a \wedge \neg z[a]$.

3.4.2 Semantics

In this section we assume we have the ingredients detailed in the previous subsection: a finite set of action labels L such that $\{R, W\} \times \mathbb{N}_n \times \mathbb{N}_3 \subseteq L$, a set of QFAs $\{\phi_\ell \mid \ell \in L\}$, and an initial state function $init$. For any $m \geq 1$, these ingredients define the protocol $\mathcal{F}(m, 3)$. The states of $\mathcal{F}(m, 3)$ are interpretations of Vars called *m-states*:

Definition 33 (m-state). *For $m \geq 1$, an m-*state* is a mapping s that interprets each $v \in \mathsf{Vars}$, with typing as follows*

$$s(v) \in \begin{cases} \mathbb{B} & \textit{if } v \in X \\ \mathbb{N}_m \cup \{\mathsf{nul}\} & \textit{if } v \in Y \\ \mathbb{N}_m \to \mathbb{B} & \textit{if } v \in Z \end{cases}$$

If s is an m-state, then s' is the interpretation of Vars' defined by $s'(v') = s(v)$ for all $v \in \mathsf{Vars}$.

Let s and t be m-states, let ϕ be a QFA, and let $j, k \in \mathbb{N}_m$ be integers. Then we write

$$s, t' \vDash \phi[j/a, k/a_1]$$

if ϕ holds when unprimed variables are interpreted by s, primed variables are interpreted by t', and a and a_1 are respectively interpreted as the constants j and k.[7] We may now define the protocol $\mathcal{F}(m, 3) = (S, \Sigma, \delta, s_0)$:

- S is the set of all m-states.
- $\Sigma = L \times \mathbb{N}_m$. In a slight abuse, we identify each $((o, p, v), j) \in \Sigma$ where $(o, p, v) \in \{R, W\} \times \mathbb{N}_n \times \mathbb{N}_3$. with the memory event $(o, p, j, v) \in \mathcal{M}$
- δ is the set of all triples $(s, (\ell, j), t')$ such that s and t are m-states, $(\ell, j) \in \Sigma$, and for all $k \in \mathbb{N}_m$ we have $s, t' \vDash \phi_\ell[j/a, k/a_1]$.
- s_0 is the unique m-state such that for each $v \in \mathsf{Vars}$ we have[8]

$$s_0(v) = \begin{cases} init(v) & \text{if } v \in X \\ \mathsf{nul} & \text{if } v \in Y \\ \lambda a.init(v) & \text{if } v \in Z \end{cases}$$

3.5 A Candidate Q

Our candidate Q is formed by a syntactic transformation on the description of $\mathcal{F}$. An important aspect of Q is the inclusion of a designated *abstract address*, denoted ξ, in the range of the non-auxiliary ARVs $Y \cup Y'$. ξ abstracts the infinite set of addresses $\mathbb{N} \setminus \mathbb{N}_n$. The states of Q are n-states with ARVs extended to include ξ in their domain.

Definition 34 (abstract state). *An* abstract state *is an m-state where $m = n$, with the exception that the range of variables in Y is extended to included ξ.*

Note that abstract states have the arrays in Z still indexed by $\mathbb{N}_n$, rather than $\mathbb{N}_n \cup \{\xi\}$. This is because the "$\xi$th" array entries will be, in effect, existentially quantified away. Thus the abstract states are a superset of the n-states, since the latter are simply the abstract states in which no $y \in Y$ is assigned ξ. An important notion in defining Q is a syntactic substitution operator $\mathsf{sub}()$, defined here.

[7] Of course this definition is somewhat informal. The formal definition of $\vDash$ would be done inductively over the syntax of Def. 32, and essentially follow the standard semantics of first-order predicate logic with equality.

[8] In defining $s_0(v)$, the expression $\lambda a.init(v)$ denotes the function that takes each address to the constant boolean value $init(v)$.

Definition 35 (substitution operator). *Given a QFA ψ, $\mathsf{sub}(\psi)$ is the QFA obtained via the following three substitutions.*

1. *each occurrence of $\alpha{=}\beta$ falling under an odd number of negations (where $\alpha, \beta \in \mathsf{ARVars}$) is replaced with*
$$(\alpha{=}\beta \wedge \neg(\alpha{=}\xi \wedge \beta{=}\xi)) \tag{3.1}$$
2. *for each $z \in Z \cup Z'$, each occurrence of $z[a]$ is replaced with $\widehat{z}$, where $\widehat{z}$ is a fresh boolean variable.*
3. *each occurrence of a remaining after the previous substitution is performed*[9] *is replaced with ξ.*

We also define $\mathsf{sub}_1(\psi)$ to be the formula obtained by performing only the first of these three substitutions.

The motivation for the first substitution is as follows. Semantically, we expect $\xi{=}\xi$ to be true, since clearly the symbol ξ is equal to itself. However, the purpose of ξ is to abstract multiple (in fact infinite) concrete addresses. In actuality we simply *don't know* if the two occurrences of ξ on either side of "$=$" abstract the same concrete address or not. It turns out that to ensure the transition relation of the abstraction Q is conservative, i.e. over-approximates the transitions of each $\mathcal{F}(m, 3)$ where $m > n$, an ARV equality test in which both operands evaluate to ξ should evaluate to false if the test falls under an odd number of negations (in the QFA). One can easily confirm that the substitutive expression (3.1) is false whenever $\alpha = \beta = \xi$ and has the truth value of $\alpha{=}\beta$ otherwise (i.e. equality tests not involving two ξs are unaffected by the substitution). This first substitution is done on *all* QFAs when defining Q.

The second and third substitutions are only used to construct the *abstract* transitions of Q, which pertain to the abstract address ξ. The second facilitates the existential quantification of the ξth entries of arrays via the introduction of fresh variables, while the third performs the necessary substitution of ξ for the "distinguished" auxiliary ARV a.

We now formally define Q to be the protocol $(S_Q, \Sigma_Q, \delta_Q, s_{0Q})$ with components as follows.

- S_Q is the set of all abstract states.

[9] Note that after the previous substitution, the only occurrences of a will be in ARV equality tests.

- $\Sigma_Q = L \times (\mathbb{N}_n \cup \{\xi\})$. As in the semantics of $\mathcal{F}(m, 3)$, we identify $((o, p, v), j) \in L \times \mathbb{N}_n$ where $(o, p, v) \in \{R, W\} \times \mathbb{N}_n \times \mathbb{N}_3$ with the memory event $(o, p, j, v) \in \mathcal{M}$. However, for the purposes of defining $\mathit{traces}(Q)$, no actions in $L \times \{\xi\}$ are considered to be memory actions.

- δ_Q is the set of all triples $(s, (\ell, j), t')$ such that s and t are abstract states, $\ell \in L$, and either

  (abstract transition) $j = \xi$ and for each $k \in \mathbb{N}_n$ we have[10]

$$s, t' \ \vDash\ \exists\widehat{zs}.\mathsf{sub}(\phi_\ell)[k/a_1] \tag{3.2}$$

  where $\widehat{zs}$ is the vector of variables $\widehat{z_1}, \ldots, \widehat{z_{|Z|}}, \widehat{z'_1}, \ldots, \widehat{z'_{|Z|}}$, or

  (concrete transition) $j \in \mathbb{N}_n$ and for each $k \in \mathbb{N}_n$ we have $s, t' \ \vDash\ \mathsf{sub}_1(\phi_\ell)[j/a, k/a_1]$.

- s_{0Q} is the initial state of $\mathcal{F}(n, 3)$.

Several modeling languages have direct support for the existential quantification in (3.2), for example TLA+ [89] and Murφ , courtesy of the ruleset construct [46]. In practice we find that only a small number of the variables $\widehat{zs}$ are actually mentioned in each ϕ_ℓ. Thus the number of distinct transitions induced by the existential quantification in (3.2) is typically much less than the worst-case exponential bound of $2^{2|Z|}$. In the later Sects. 3.7.1 and 3.7.2 we give examples of the abstraction used to define Q.

We conclude this section by proving Theorem 13, which asserts that $\mathit{traces}(Q)$ over-approximates a projected version of $\mathit{traces}(\mathcal{F}(m, 3))$ for all $m > n$, i.e. Q satisfies condition 2 of Theorem 12. To prove Theorem 13, we show that there is a natural abstraction function that takes the concrete states of $\mathcal{F}(m, 3)$ to abstract states of Q. Furthermore, the transitions of Q are conservative with respect to this abstraction function. Though the proof of Theorem 13 is conceptually straightforward, careful consideration of the multitude of base-cases in the proofs of the supporting Lemmas 30 and 31 causes them to be somewhat tedious.

Theorem 13. *$\mathit{traces}(\mathcal{F}(m, 3)) \restriction \mathcal{M}(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3) \subseteq \mathit{traces}(Q)$ for all $m > n$.*

Proof. Immediate from Lemmas 32 and 29 below. □

[10] Here the existential quantification of boolean variables has the usual semantics that $\exists x.\psi$ is equivalent to $\psi[\mathsf{T}/x] \vee \psi[\mathsf{F}/x]$.

Lemmas 32 and 29 make use of the notion of *simulation relation*, which is an association between states of two different systems used to prove that one system in some sense over-approximates the other. Simulation is a common theme across many papers on formal verification, and appears in many guises and variants. The literature tends to attribute the origin of the idea to Milner [100]. Simulation relations often imply certain language containments; ours being no exception. Here we craft a notion of simulation relation that suits our needs.

Definition 36 (Σ-simulation). *Given two automata*[11] $A_1 = (S_1, \Sigma_1, \delta_1, s_{01})$ *and* $A_2 = (S_2, \Sigma_2, \delta_1, s_{02})$ *and an alphabet* Σ*, a relation* $R \subseteq S_1 \times S_2$ *is said to be a* Σ-simulation relation *from* A_1 *to* A_2 *if we have both of the following.*

1. *for all* $(s_1, s_2) \in R$*, for each transition* $(s_1, \alpha_1, s'_1) \in \delta_1$*, there exists* s'_2 *such that* $(s_2, \alpha_2, s'_2) \in \delta_2$ *and* $(s'_1, s'_2) \in R$*. Furthermore, we require that* $\alpha_1 \in \Sigma$ *implies* $\alpha_2 = \alpha_1$*, while* $\alpha_1 \notin \Sigma$ *implies* $\alpha_2 \notin \Sigma$*.*

2. $(s_{01}, s_{02}) \in R$*.*

If there exists a Σ*-simulation relation from* A_1 *to* A_2 *we say that* A_2 Σ-simulates A_1*.*

Intuitively, we can think of Σ as specifying a set of *observable* symbols. Whenever A_1 can transition on an observable, A_2 must be able to transition on the same observable. Also, whenever A_1 can transition on an unobservable, then A_2 must be able to transition on *some* unobservable, not necessarily the same one. A straightforward lemma relating Σ-simulation to projected language containment is the following.

Lemma 29. *If automaton* A_2 Σ*-simulates automaton* A_1 *then* $\mathcal{L}(A_1) \uparrow \Sigma \subseteq \mathcal{L}(A_2) \uparrow \Sigma$*.*

Proof. Let R be a Σ-simulation relation from A_1 to A_2, and let $\tau = \tau(1) \dots \tau(m)$ be a string in $\mathcal{L}(A_1) \uparrow \Sigma$. Then there exists a string $\rho_1 = \rho_1(1) \dots \rho_1(k)$ of length $k \geq m$ such that $\rho_1 \in \mathcal{L}(A_1)$ and $\rho_1 \uparrow \Sigma = \tau$. It follows that there exists a sequence of states $s_1^0, \dots, s_1^k$ of A_1 such that s_1^0 is the initial state of A_1, and for each $1 \leq i \leq k$, we have that $(s_1^{i-1}, \rho_1(i), s_1^i)$ is a transition of A_1. From Def. 36, there exists a string $\rho_2 = \rho_2(1) \dots \rho_2(k)$ and a sequence of states $s_2^0, \dots, s_2^k$ of A_2 such that

[11] Restricting attention to automata having all states accepting simplifies this definition. Furthermore, we only apply Σ-simulation to protocols (in which all states are inherently "accepting"). Hence we have intentionally neglected to specify the accepting states (normally the fifth component) for these automata.

- s_2^0 is the initial state of A_2, and
- for each $1 \leq i \leq k$, we have that $(s_2^{i-1}, \rho_2(i), s_2^i)$ is a transition of A_2, and
- for each $0 \leq i \leq k$, we have that $(s_1^i, s_2^i) \in R$, and
- $\rho_2 \uparrow \Sigma = \tau$

Hence $\rho_2 \in \mathcal{L}(A_2)$ and $\tau \in \mathcal{L}(A_2) \uparrow \Sigma$. □

We now define a function H that, for any fixed $m > n$, takes the set of m-states to the set of abstract states, and then we prove that H is in fact a $\mathcal{M}(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3)$-simulation relation from $\mathcal{F}(m,3)$ to Q. For any $j \in \mathbb{N} \cup \{\mathsf{nul}\}$, we define $\widetilde{j} = \xi$ if $j \in \mathbb{N} \setminus \mathbb{N}_n$, otherwise we define $\widetilde{j} = j$.

Definition 37 (H). *Let S be the set of m-states for some fixed $m > n$. Define the function $H : S \to S_Q$ such that for each $s \in S$ we have*

1. *for each $x \in X$, $H(s)(x) = s(x)$*
2. *for each $y \in Y$, $H(s)(y) = \widetilde{s(y)}$*
3. *for each $z \in Z$, $H(s)(z)$ is the restriction of $s(z)$ to the domain $\mathbb{N}_n$.*

Before stating that H is a simulation relation below in Lemma 32, we first present the two supporting Lemmas 30 and 31. Lemma 30 states that a transition of $\mathcal{F}(m,3)$ on address $j \leq n$ is simulated by a concrete transition of Q, while Lemma 31 states that a transition on an address $j > n$ is simulated by an abstract transition of Q.

Lemma 30. *For any QFA ϕ, m-states s and t, and $j, k \in \mathbb{N}_n$,*

$$s, t' \vDash \phi[j/a, k/a_1] \tag{3.3}$$

implies

$$H(s), H(t)' \vDash \mathsf{sub}_1(\phi)[j/a, k/a_1] \tag{3.4}$$

Proof. Without loss of generality, let us assume that ϕ is written in negation normal form.[12] The proof is by induction on the structure of ϕ. The base cases correspond to ϕ being an atom or the negation of an atom. When the atom is an ARV equality test, the argument for the non-negated and negated variants are nonanalogous; thus these are handled separately below. All other negated atoms are not considered because their proofs are analogous to that of their non-negated versions. We also omit treatment of the primed variable versions, again because of analogy. As usual, $x \in X$, $y, w \in Y$, and $z \in Z$.

1. $\phi = x$. Trivial, since H preserves the values of the X variables.

2. $\phi = z[a]$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = \phi[j/a, k/a_1] = z[j]$, thus (3.3) implies (3.4).

3. $\phi = z[a_1]$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = \phi[j/a, k/a_1] = z[k]$, thus (3.3) implies (3.4).

4. $\phi = y{=}\mathsf{nul}$. Trivial, since H preserves equality with nul.

5. $\phi = y{=}a$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = \phi[j/a, k/a_1] = y{=}j$, thus (3.3) implies that $s(y) = j$, and thus $H(s)(y) = j$, yielding (3.4).

6. $\phi = \neg y{=}a$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = \neg\,(y{=}j \wedge \neg\,(y{=}\xi \wedge j{=}\xi))$, which, since $\xi \neq j$, reduces to $\neg y{=}j$. Thus (3.3) implies $s(y) \in \mathbb{N}_m \setminus \{j\}$, and it follows that $H(s)(y) \in \{\xi\} \cup (\mathbb{N}_n \setminus \{j\})$ and thus (3.4).

7. $\phi = y{=}a_1$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = y{=}k$, and $\phi[j/a, k/a_1] = y{=}k$. Thus (3.3) implies $s(y) = k \leq n$, and hence $H(s)(y) = k$, yielding (3.4).

8. $\phi = \neg y{=}a_1$. Similar to item 6.

9. $\phi = a{=}a_1$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = \phi[j/a, k/a_1] = j{=}k$. Thus (3.3) implies (3.4).

10. $\phi = \neg a{=}a_1$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = \neg\,(j{=}k \wedge \neg\,(j{=}\xi \wedge k{=}\xi))$, which reduces to $\neg j{=}k$, and $\phi[j/a, k/a_1] = \neg j{=}k$. Thus (3.3) implies (3.4).

11. $\phi = y{=}w$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = \phi[j/a, k/a_1] = y{=}w$. Thus (3.3) implies that $s(y) = s(w)$, which always implies $H(s)(y) = H(s)(w)$, and hence (3.4).

[12] A QFA is in *negation normal form* if negations are only applied to atoms. Any QFA can be rewritten in negation normal form by pushing negations inwards using De Morgan's law.

12. $\phi = \neg y{=}w$. Then $\mathsf{sub}_1(\phi)[j/a, k/a_1] = \neg\,(y{=}w \wedge \neg\,(y{=}\xi \wedge w{=}\xi))$ and $\phi[j/a, k/a_1] = \neg y{=}w$. Assuming (3.3), we case split depending on the propositions $s(y) \leq n$ and $s(w) \leq n$.
    - $s(y) \leq n$ and $s(w) \leq n$. Then $H(s)(y) = s(y) \neq H(s)(w) = s(w)$ and thus (3.4).
    - $s(y) > n$ and $s(w) \leq n$. Then $H(s)(y) = \xi \neq H(s)(w)$ and thus (3.4).
    - $s(y) \leq n$ and $s(w) < n$. Analogous to the previous sub-case.
    - $s(y) > n$ and $s(w) > n$. Then $H(s)(y) = H(s)(w) = \xi$, and again we have (3.4).

For the inductive step, note that if $\phi = \phi_0 \wedge \phi_1$ and (3.3) holds of ϕ, then clearly (3.3) holds of both ϕ_0 and ϕ_1. By our inductive hypothesis (3.4) holds of both ϕ_0 and ϕ_1, and hence also ϕ. The inductive step for disjunction is analogous. □

Lemma 31. *For any QFA ϕ, m-states s and t, integer $j \in \mathbb{N}_m \setminus \mathbb{N}_n$, and $k \in \mathbb{N}_n$,*

$$s, t' \vDash \phi[j/a, k/a_1] \tag{3.5}$$

implies

$$H(s), H(t)' \vDash \exists \widehat{zs}.\mathsf{sub}(\phi)[k/a_1] \tag{3.6}$$

Proof. For any m-state u, let $H^j(u)$ be the interpretation of the variables $X \cup Y \cup Z \cup \widehat{zs}$ that is equivalent to $H(u)$ for the variables $X \cup Y \cup Z$, and, for each $\widehat{z} \in \widehat{zs}$, we set $H^j(u)(\widehat{z}) = u(z)(j)$. We prove that for any $k \in \mathbb{N}_n$, we have

$$H^j(s), H^j(t)' \vDash \mathsf{sub}(\phi)[k/a_1] \tag{3.7}$$

which clearly implies (3.6). The proof is by induction on the structure of ϕ, and, as in the proof of Lemma 30, we assume that ϕ is in negation normal form. Many of the base cases are very similar to cases of the proof of Lemma 30. Here we only treat those that have a significant difference; the cases are numbered according to the corresponding cases of Lemma 30. As usual, $x \in X$, $y, w \in Y$, and $z \in Z$.

2. $\phi = z[a]$. Then $\mathsf{sub}(\phi)[k/a_1] = \widehat{z}$, and $\phi[j/a, k/a_1] = z[j]$, thus (3.5) implies (3.7).

6. $\phi = y{=}a$. Then $\mathsf{sub}(\phi)[k/a_1] = y{=}\xi$, and $\phi[j/a, k/a_1] = y{=}j$. Thus, $s(y) = j > n$, and hence $H^j(s)(y) = \xi$, yielding (3.7).

7. $\phi = \neg y{=}a$. Then $\mathsf{sub}(\phi)[k/a_1] = \neg\,(y{=}\xi \wedge \neg\,(y{=}\xi \wedge \xi{=}\xi))$, which is a tautology, thus we have (3.7).

9. $\phi = \neg y{=}a_1$. Then $\mathsf{sub}(\phi)[k/a_1] = \neg\,(y{=}k \wedge \neg\,(y{=}\xi \wedge k{=}\xi))$, which is semantically equivalent to $\neg y{=}k$. Also, $\phi[j/a, k/a_1] = \neg y{=}k$, thus (3.5) implies that $s(y) \neq k$, which implies that $H^j(s)(y) \neq k$, and hence (3.7).

10. $\phi = a{=}a_1$. Then $\phi[j/a, k/a_1] = j{=}k$, and therefore (3.5) cannot hold, since $k \leq n < j$.

11. $\phi = \neg a{=}a_1$. Then $\mathsf{sub}(\phi)[k/a_1] = \neg\,(\xi{=}k \wedge \neg\,(\xi{=}\xi \wedge k{=}\xi))$, which is a tautology. Hence (3.7).

The inductive step is analogous to that of the proof of Lemma 30. □

Lemma 32. *H is a $\mathcal{M}(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3)$-simulation relation from $\mathcal{F}(m, 3)$ to Q.*

Proof. First, we note that condition 2 of Def. 36 is satisfied, since $H(s_0) = s_{0Q}$, where s_0 is the initial state of $\mathcal{F}(m, 3)$. The remainder of the proof is devoted to condition 1 of Def. 36. Let s and t be m-states, and $j \in \mathbb{N}_m$. We will show that if $(s, (\ell, j), t')$ is a transition of $\mathcal{F}(m, 3)$, then $(H(s), (\ell, \widetilde{j}), H(t)')$ is a transition of Q. This implies the lemma because $(\ell, j) \in \mathcal{M}(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3)$ iff $(\ell, \widetilde{j}) \in \mathcal{M}(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3)$, and if $j \in \mathbb{N}_n$ then $\widetilde{j} = j$. Now, assuming $(s, (\ell, j), t')$ is a transition of $\mathcal{F}(m, 3)$, then for all $k \in \mathbb{N}_m$ we have $s, t' \vDash \phi_\ell[j/a, k/a_1]$; in particular this is true for all $k \in \mathbb{N}_n$, since $n < m$. We case-split on whether $j \leq n$. If $j \leq n$, then by Lemma 30 we have $H(s), H(t)' \vDash \mathsf{sub}_1(\phi_\ell)[j/a, k/a_1]$ for all $k \in \mathbb{N}_n$, and thus $(H(s), (\ell, j), H(t)') \in \delta_Q$. Now suppose that $j > n$. Then, by Lemma 31, we have that $H(s), H(t)' \vDash \exists \widehat{zs}.\mathsf{sub}(\phi_\ell)[k/a_1]$ for all $k \in \mathbb{N}_n$, and thus $(H(s), (\ell, \xi), H(t)') \in \delta_Q$. Note that neither (ℓ, j) nor (ℓ, ξ) are in $\mathcal{M}(\mathbb{N}_n, \mathbb{N}_n, \mathbb{N}_3)$, thus Def. 36 is satisfied in this case. □

3.6 Experimental Results

To evaluate our technique, we experimented with two protocols, which we call PIR and WIS. Both of these protocols are SSC but not serial, hence trace reordering requirements are nontrivial. PIR is a simplified abstraction of the Piranha protocol [13] as presented by Qadeer [113]. WIS is a simplification of a directory based protocol with Scheurich's optimization [116]. Our implementation

is based on, but is simpler than, the description verified by Braun et al. [22], which was designed by the University of Wisconsin Multifacet group. We discuss aspects of both protocols here and describe the ways in which our implementation of WIS is simpler than the version used by Braun et al. [22].

In both PIR and WIS, each processor p has a cache that contains, for certain memory locations j, a data value $data[p][j]$ and an access permission $access[p][j]$. The permission may be modifiable (M), shared (S), or invalid (I). Processor p may read the data value for location j if $access[p][j]$ is S or M, and may write (change) the data value if $access[p][j]$ is M.

In addition to the caches, PIR has an incoming response queue for each processor and an owner processor for each memory location. An invariant of PIR is that the owner of location j is either some processor p for which $access[p][j] \in \{\mathsf{S}, \mathsf{M}\}$, or the owner is null. Requests by a processor to change its access level or to get a data value for a location are not modeled explicitly. Rather, if $owner[j] = p$, in one transition of the protocol an exclusive request acknowledgment message $(\mathsf{ACKX}, data[p][j], j)$ may be placed on the queue of any processor $p' \neq p$, in which case an invalidate message (INV, j) is placed on the queues of any processor p'' (other than p and p') with $access[p''][j] \neq \mathsf{I}$. Also, $owner[j]$ is set to null and $access[p][j]$ is set to I. In a later transition, when triple (ACKX, d, j) is at the head of p''s queue, the triple may be removed from the queue, in which case $data[p'][j]$ is set to d, $access[p'][j]$ is set to M, and $owner[j]$ is set to p'. Also, if (INV, j) is at the head of the queue of p'', the message may be removed from the head of the queue, in which case $access[p''][j]$ is set to I. Other access permission changes are modeled similarly. The state space of PIR is relatively small because requests of processors are not modeled explicitly, and moreover, no directory is used to store the set of processors with shared access to a memory location. However, the use of queues ensures that the traces of PIR are interesting in the sense that they are SSC but not serial.

The WIS protocol contains many details that are not present in PIR. Each processor and the directory has both a request *and* a response input queue, with different uses. For example, invalidate messages to processors are placed on the request input queue, whereas messages containing data values are placed on the response input queue. A directory maintains information regarding which processors have exclusive or shared access to a memory location. Requests for exclusive permissions received by the directory are redirected to the current owner if it is owned with permissions M.

Because of the multiple queues, several race conditions can arise. For example, the directory might receive a request from processor p' to have exclusive access to location j after exclusive access was granted to another processor p. It is possible that p receives the request to send j's data value to p' before p has even received the message granting exclusive access, which also contains the data value for j. As a result, a processor has 9 permission types per location in its cache in addition to the I ,S, and M types. For example, $access[p][j]$ may be IMS, indicating that p requested M access when $access[p][j]$ was I, p has not yet transitioned to M access, but already another request to p to downgrade to S access has been received. To keep the state space within manageable proportions, our implementation of the WIS protocol does not model a certain queue of Braun et al. [22]. This queue is used for transmission of data within a processor, rather than between processors, and its removal does not significantly change the protocol.

Both PIR and WIS were modeled using the Murφ language and involved 165 and 1397 lines of code, respectively (excluding comments). The Murφ language, of course, is more expressive than the formalism of Sect. 3.4, but the implementations only involve constructs that conform to this formalism. Although the construction in Sect. 3.5 is obviously automatable, we do not yet have an implementation for the Murφ language, so we performed the construction by hand, following the described syntactic transformations. In many instances we applied obvious optimizations to the code resulting from the syntactic transformation of Sect. 3.5; in all cases we believe these optimizations could be performed algorithmically.

For WIS, the model checking runs against the "automatically"-generated Q spaced out, so the protocol was manually abstracted further, yielding Q'; the numbers in Table 3.1 refer to the completed verification runs against Q'. The essential difference between Q and Q' is that in the latter, all fields (except for the *address* field) in messages pertaining to the abstracted address ξ are abstracted.[13] The same approach was taken with a local record at each processor called the *transaction buffer*. Since Q' is an over-approximation of Q, the fact that the verification succeeded for Q' proves that verification would have succeeded for Q, *had we had sufficient memory resources*. In other words, our automatic construction of Q was accurate enough to prove SSC of the protocol family. In general, one could envision a tool that automatically attempts such additional abstractions

[13]Here, by *abstracted*, we mean that certain state variables v are, on occasion, assigned a "no-information" value (akin to $\perp$ discussed in Sect. 3.7.1). Whenever v is compared to some constant, the code is changed so that if v is the no-information value, then the comparison succeeds.

Protocol	k	#states	time	depth	prob
PIR $(q = 1)$	1	49365	7	18	≈ 0
	2	138621	9	20	≈ 0
PIR $(q = 2)$	1	3782880	100	21	≈ 0
	2	10558306	278	23	≈ 0
PIR3$(q = 1)$	1	125865495	9244	36	≤ 0.000024
	2	374557312	25640	38	≤ 0.021101
WIS $(q = 1)$	1	171088424	49660	53	≤ 0.000013
	2	375967684	110211	59	≤ 0.000324

Table 3.1: Results for the PIR (Piranha) and WIS (Wisconsin Directory) Protocols. All runs are with the number of processors $n = 2$, except for PIR3 with $n = 3$. We experimented with two versions of PIR: with processors' queue depth $q = 1$ and $q = 2$. We use a method of Qadeer [113] to prove SSC of our generated finite-state abstract protocol Q. For each $k \in \mathbb{N}_n$, Qadeer's method requires a separate model-checking run that checks for the presence of k-nice cycles. Times are in seconds, on a 2Ghz Intel Xeon with 4GB memory running Linux. "prob" is an upper-bound on the probability of missing states due to hash compaction, as reported by Murφ [121] (40 bit hashes for PIR; 42 bits for WIS). The $n = 2$ runs concluded successfully, enabling us to conclude SSC for each example over all address counts m and data value counts v where $m > n = 2$. The PIR3 results are a preliminary attempt with $n = 3$ processors. For PIR3 with $k = 2$, we used 35-bit hashes. We have not yet completed a $k = 3$ run at press time.

if model checking the original Q is intractable.

Table 3.1 details our experimental verification efforts for three configurations of PIR and one configuration of WIS. The 2-processor runs were successful, showing that our generated abstraction is accurate enough, even for these protocols with non-trivial reordering properties (which are therefore hard-to-prove sequentially consistent).

3.7 Comments on our Abstraction

In this section we consider two points of interest related to our abstraction of $\mathcal{F}$ into Q. First we defend our choice by showing that a better-known and perhaps simpler abstraction can be too coarse, while ours succeeds. Second, we observe that though our abstraction is successful against the two example protocols in Sect. 3.6, it is capable of failing because of its inherent incompleteness; this deficiency is typical of abstraction-based verification approaches. We provide a simple and reasonable protocol for which our approach fails, and we suggest a possible solution to this problem.

3.7.1 Comparison to Other Abstractions

Similar abstraction approaches such as those of McMillan [95, 96] and Chou et al. [28], differ in a subtle but important way from ours.[14] In effect, they use three-valued logic to define the abstracted system. Any reference to an abstracted array entry $z[\xi]$ evaluates to a "no-information" logic value $\perp$. Equality comparisons with $\perp$ always yield $\perp$, and boolean connectives involving $\perp$ are evaluated according to the usual three-valued logic rules.[15] If a transition between two states evaluates to T *or* $\perp$, then the transition is admitted by the abstraction. For the purposes of this discussion, we will call this approach $\perp$-*abstraction*, while our approach will be called $\exists$-*abstraction*.

The difference between $\perp$-abstraction and $\exists$-abstraction is that the former, in general, is coarser. One can easily construct an example to illustrate this; suppose our system has a QFA

$$(z[a] \wedge \neg z[a]) \wedge \psi \tag{3.8}$$

where ψ is some constraint on the primed variables. This transition is guarded by $z[a] \wedge \neg z[a]$. Using $\exists$-abstraction, we (essentially) get the abstract transition

$$\exists \widehat{z}.(\widehat{z} \wedge \neg \widehat{z}) \wedge \mathsf{sub}(\psi) \tag{3.9}$$

The guard of (3.9) always evaluates to F, as does that of the concrete transition (3.8), therefore neither transition can ever occur. However, if we apply $\perp$-abstraction, the guard of the transition becomes $\perp \wedge \neg\perp$, which evaluates to $\perp$ in the three-valued logic. This, in effect, means that the transition becomes unguarded.[16] Hence the transition, which cannot fire in either of the concrete system or the $\exists$-abstraction, is *always enabled* in the $\perp$-abstraction, and thus $\perp$-abstraction is strictly coarser.

In general, $\exists$-abstraction will likely be finer for rules that syntactically mention the abstracted entry of the same array multiple times. In such circumstances, $\exists$-abstraction ensures that all occurrences are considered to have the same value, while $\perp$-abstraction makes no such guarantee. Of course, one would not expect to see syntactic constructs resembling the guard of (3.8) in real

[14] Of course, they differ in the not-so-subtle fact that they abstract processes rather than addresses in their applications.

[15] Which are defined by $\mathsf{T} \wedge \perp \equiv \perp \wedge \perp \equiv \mathsf{F} \vee \perp \equiv \perp \vee \perp \equiv \neg\perp \equiv \perp$, $\mathsf{F} \wedge \perp \equiv \mathsf{F}$, and $\mathsf{T} \vee \perp \equiv \mathsf{T}$

[16] To see this, note that $(\perp \wedge \neg\perp) \wedge \psi \;\equiv\; (\perp \wedge \perp) \wedge \psi \;\equiv\; \perp \wedge \psi$, which is either $\perp$ or F, depending on the value of ψ. Therefore, as long as the next state satisfies the constraints imposed by ψ, the transition is admitted independently of the current state.

Guard	Command
$buffer.msgType{=}\mathsf{GETX}$	$SendGETX(a, dir[a].owner);$
$\wedge\ buffer.address{=}a$	$buffer := \mathsf{empty};$
$\wedge\ dir[a].state{=}\mathsf{M}$	$dir[a].owner := buffer.processor;$
$\wedge\ \neg requestQfull(dir[a].owner)$	$dir[a].state := \mathsf{MtoM}$

(a) The unabstracted QFA

$\exists\widehat{do} \in \{1, 2\}$.

Guard	Command
$buffer.msgType{=}\mathsf{GETX}$	$SendGETX(\xi, \widehat{do})$
$\wedge\ buffer.address{=}\xi$	$buffer := \mathsf{empty}$
$\wedge\ \neg requestQfull(\widehat{do})$	

(b) Our $\exists$-abstraction of the QFA

Guard	Command
$buffer.msgType{=}\mathsf{GETX}$	$(SendGETX(\xi, 1) \oplus SendGETX(\xi, 1));$
$\wedge\ buffer.address{=}\xi$	$buffer := \mathsf{empty}$
$\wedge\ (\neg requestQfull(1) \vee \neg requestQfull(2))$	

(c) A $\perp$-abstraction of the QFA

Figure 3.3: Example of a QFA (expressed in guarded commands) for which $\perp$-abstraction is too coarse

descriptions. However, more elaborate and subtle examples in which $\perp$-abstraction is too coarse exist. Fig. 3.3 gives an example of a QFA, based on a rule appearing in the Wisconsin directory protocol (one of the experimental protocols of Sect. 3.6), along with its $\exists$-abstracted and $\perp$-abstracted versions. For this QFA, in contrast with $\exists$-abstraction, $\perp$-abstraction is *too coarse* for a typical verification goal to succeed.

Fig. 3.3(a) is the unabstracted QFA, expressed in guarded commands.[17] In this example, we assume that there are $n = 2$ processors with names 1 and 2. Roughly, this QFA models a protocol transition occurring when the directory receives a request for modify (M) access to an address a, but the directory has previously granted modify access to another processor (called the *owner*). The directory handles the request by forwarding it to the owner, if possible. The variable $buffer$ stores the message incoming to the directory. The name of the current owner of address a is stored in $dir[a].owner$. The guard ensures that the owner has room in its incoming request queue to receive

[17]The correspondence between our guarded commands and the AVPPF formalism described in Sect. 3.4 is straightforward. Roughly, to convert a guarded command into a QFA, one primes all the left-hand sides of assignments, adds constraints fixing all variables that are left unassigned, and conjoins the resulting expression with the guard.

a message. The command, among other things, actually sends the message to $dir[a].owner$, via the macro $SendGETX(a, p)$, which appends a message to p's queue requesting that it relinquish M access to address a.

Fig. 3.3(b) depicts the result of applying our ∃-abstraction to the QFA of Fig. 3.3(a), after some modest simplifications. Note that the two updates to array entries at the ξ index (i.e. the last two lines of the command of Fig. 3.3(a)) vanish as a result of the existential quantification of the "primed and hatted" variables corresponding to this state. However, since the unprimed array entry $dir[a].owner$ is mentioned more than once, we must keep the quantification. We use $\widehat{do}$ for the hatted variable substituted in for $dir[a].owner$ as a result of the second substitution of Def. 35. Note that even though $\widehat{do}$ can nondeterministically be either of 1 or 2, from the scope of the quantification we see that the processor that is confirmed (by the guard) to have room in its request queue is the *same* processor that is actually sent the message in the command.

$\perp$-abstraction can be performed on the QFA of Fig. 3.3(a) in several equivalent ways, one of which is shown in Fig. 3.3(c). The boolean exclusive-OR $\oplus$ in the command enforces that exactly one of the two processors is forwarded the request message. The crux of the matter is that the processor confirmed to have room in its request queue can differ from the processor that the message is actually sent to. This can clearly be problematic if the latter has a full queue. For instance, depending on how the macro $SendGETX(a, p)$ is "implemented", an earlier message lingering in p's queue could be overwritten when the $\perp$-abstracted QFA fires.

3.7.2 Example of Verification Failure

Fig. 3.4 describes a AVPPF $\mathcal{C}$ that is SSC (in fact it is serial) for which the verification approach of this chapter fails. Though this AVPPF was concocted by the author, it is by no means unrealistic. Fig. 3.4(a) specifies the transitions of $\mathcal{C}$ using guarded commands and Fig. 3.4(b) defines the initial state predicate. As specified in Fig. 3.4 the description is parameterized by the number of data values, addresses, *and* processors. However, in keeping with the theme of the chapter, we consider the number of processors to be fixed at some n. Fig. 3.4(c) gives the architecture of $\mathcal{C}$ (when $n = 2$) annotated with the state variables.

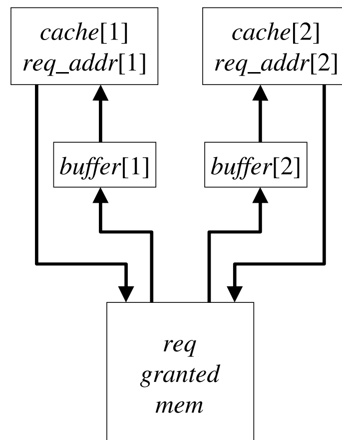
Behaviorally, the AVPPF is relatively uncomplicated, here we give only a brief description. Each processor p has a cache, and each address a in each cache has an associated $valid$ bit; when

Action Label	Guard	Command
(R, p, a, d)	$cache[p][a].valid$ $\wedge cache[p][a].data = d$	
(W, p, a, d)	$cache[p][a].valid$	$cache[p][a].data := d$
$Request(p, a)$	$\neg cache[p][a].valid$ $\wedge req_addr[p] = \mathsf{nul}$	$req[a][p] := \mathsf{T}$; $req_addr[p] := a$
$WriteBack(p, a)$	$cache[p][a].valid$	$cache[p][a].valid := \mathsf{F}$; $mem[a] := cache[p][a].data$; $granted[a] := \mathsf{F}$
$Grant(p, a)$	$\neg granted[a]$ $\wedge req[a][p]$	$granted[a] := \mathsf{T}$; $req[a][p] := \mathsf{F}$; $buffer[p] := mem[a]$
$ReceiveGrant(p, a)$	$req_addr[p] = a$ $\wedge buffer[p] \neq \mathsf{empty}$	$cache[p][a].data := buffer[p]$; $cache[p][a].valid := \mathsf{T}$; $req_addr[p] := \mathsf{nul}$; $buffer[p] := \mathsf{empty}$

(a) Transitions of $\mathcal{C}$

$$\forall a, p. \begin{pmatrix} \neg cache[p][a].valid \\ \wedge \neg req[a][p] \\ \wedge\ req_addr[p] = \mathsf{nul} \\ \wedge \neg granted[p] \\ \wedge\ buffer[p] = \mathsf{empty} \end{pmatrix}$$

(b) Initial state predicate of $\mathcal{C}$. Note that this predicate can easily be translated into an initial state function $init$ as required by our formalism.



(c) Architecture of $\mathcal{C}$ instantiated with 2 processors

Figure 3.4: Example AVPPF $\mathcal{C}$ for which our $\exists$-abstraction is too coarse to verify SSC

this bit is set, we will say that p has *access* to a. When a processor has access to an address, it is guaranteed to be the only such processor, and is hence permitted to both read and write the address. A central directory coordinates access requests. When processor p wants access to an address a, it makes a record of the address locally (in $req_addr[p]$), and a bit recording the request is set in the directory ($req[a][p]$); note that a processor can have at most one outstanding request at any time. Using the bit $granted[a]$, the directory remembers if *some* processor has access to address a. If not, it can nondeterministically choose a processor p that has previously requested access to a, and grant access to p. This is achieved by writing the data value the directory holds for a into a buffer for p. When p sees that there is data in its buffer, it "knows" the relevant address by reading its local variable $req_addr[p]$. It may then clear $req_addr[p]$, empty its buffer, and update its cache appropriately. Note that the usage of $req_addr[p]$ eliminates the need for an address field in $buffer[p]$.

To see that our approach fails to verify $\mathcal{C}$, consider the abstracted version of the action $Grant$ occurring in Q, call it $absGrant$. Translated into our logic and simplified, $absGrant$ looks like

$$absGrant \quad = \quad \exists \widehat{mem}.buffer[p]'{=}\widehat{mem} \wedge \psi_{\mathit{fix}} \tag{3.10}$$

where ψ_{fix} simply expresses that all primed state variables other than $buffer[p]'$ are left fixed. In particular, (3.10) has no guard (i.e. constraint on unprimed variables), and hence is always enabled. If processor p has previously requested access to a non-abstract address j, i.e. $req_addr[p] = j$, firing $absGrant$ will result in p "thinking" that it has received data (and thus access) for address j. The data value $\widehat{mem}$ introduced by the existential quantifier in (3.10) can clearly be a value that has never been written to address j, thus SSC will be violated when p subsequently reads address j.

How then can we handle AVPPFs for which our abstraction is too coarse, such as $\mathcal{C}$? One possible solution is to require the user to provide so-called *noninterference lemmas* [28, 95, 96]. A noninterference lemma is an invariant used to refine the abstraction when verification fails as a result of an abstracted component provoking erroneous behavior in a concrete component. Deploying a noninterference lemma involves both proving that the lemma is an invariant and restricting the behavior of an abstracted component according to the lemma.

In our case, the abstracted component corresponds to all the ξth entries of arrays, and the interference stems from the fact that the existentially quantified state of such entries can behave as if a

processor has an outstanding access request for the address ξ, when in fact it does not. A noninterference lemma can be surmised by contemplating the question of why this interference doesn't arise in the concrete protocol. The answer: an invariant of the AVPPF is that a processor p is only ever sent data for the address named in $req_addr[p]$. Further reasoning leads us to the following noninterference lemma, where p and a respectively range over processors and addresses.

$$\forall a, p.(req[a][p] \rightarrow req_addr[p]{=}a) \tag{3.11}$$

To refine our abstraction, we may strengthen any abstract transition with the noninterference lemma (3.11). Doing so to the guard of $Grant(p, a)$ (and applying universal instantiation and modus ponens) yields the strengthened guard:

$$\neg granted[a] \wedge req[a][p] \wedge req_addr[p]{=}a \tag{3.12}$$

Now using our $\exists$-abstraction on $Grant$ with the guard strengthened to (3.12) gives us a refined $absGrant$:

$$absGrant' \;\; = \;\; \exists \widehat{mem}.req_addr[p]{=}\xi \wedge buffer[p]'{=}\widehat{mem} \wedge \psi_{\mathit{fix}} \tag{3.13}$$

The reader may confirm that the verification failure induced by using (3.10) as the abstraction of $Grant$ is abolished by instead using (3.13).

Of course, if we apply a noninterference lemma such as (3.11), we are obligated to prove it. This can be done using the surprising and powerful result that *a noninterference lemma can be proven in the abstracted system that has been refined by the very same lemma we aim to prove* [28, 95, 96]. In our example, this equates to verifying that (3.11) holds (for a ranging over concrete addresses only) in the abstraction that has been refined with $absGrant'$.

Chapter 4

Process Parameterized Verification using BDDs

4.1 Introduction

This chapter develops an approach for performing automatic verification of protocols parameterized by processors. The technique is applicable to a certain class of systems involving an arbitrary number of interacting finite state clients. Since these clients do not necessarily model processors, we will use the more general term *process* to refer to a client. Because of the difficulty inherent in this problem, we turn attention away from the verification of sequential consistency. Instead we handle only assertional verification, wherein one characterizes the "bad" states and strives to verify that no such states are reachable in the system.

To motivate the approach of this chapter, consider the following simple verification strategy. First model check the system restricted to one process, then two processes, then three, etc.; if a bug is found, then terminate the procedure. The obvious drawback is that if no bug is found, in general there is no way of determining if the system truly is bug-free, or if we simply have not analyzed enough processes for a bug to manifest itself. The situation is analogous to an attempt to solve the halting problem for a Turing machines by exploring computations with ever increasing time bounds; there is no computable "cut-off" bound at which one can conclude that if the Turing machine has not yet halted, then it never will. This analogy between the halting problem and parameterized verification has been formalized by Apt and Kozen [10], their result being that the latter, like the

former, is undecidable.

Our primary contribution is the definition of a class of systems equipped with a *Convergence Theorem* which allows us to dynamically determine when we have “seen enough” to conclude correctness in the above verification strategy. Our class of systems, called *nicely sliceable WSTS*, is contained in the class of *well-structured transition systems* (WSTSs) [2,56,57] and our development borrows heavily from WSTS theory. The problem we solve via our approach is known to be decidable via the previous work on WSTS. The advantage we claim over the classical algorithm is empirical; since our approach involves model-checks on only finite state truncations of the full unbounded (infinite-state) parameterized system, we can employ BDD-based symbolic model checking [23,24]. BDDs are well-known to have the ability to represent large state spaces succinctly, which we believe to give our algorithm an advantage when processes have large “local” state spaces. We note that though our nicely sliceable WSTSs encompass many types of infinite state systems [20], in this chapter we focus on *petri nets* and the richer *broadcast protocols*, which both model systems composed of an arbitrary number of communicating finite-state processes. Broadcast protocols in particular can model certain simple shared memory protocols.

For nicely sliceable WSTSs, our algorithm is both a sound and complete verification method. There are several system attributes possessed by common shared memory protocols that are precluded by broadcast protocols. One of these is the so-called *conjunctive guard* [49, 51], which refers to a transition in a parameterized system of processes that may only occur if the local state of *each* process satisfies some predicate. To further extend the applicability of our algorithm, we introduce a new reduction that soundly converts the verification of parameterized systems with conjunctive guards into a verification problem on broadcast protocols. The reduction is proved to be complete if a certain decidable side condition holds. This allows us to access industrially relevant challenge problems from parameterized memory system verification. Our empirical results show that, although our new method performs worse than the classical approach on small petri net examples, it is capable of efficiently verifying systems with large local state. The classical WSTS algorithm is either much slower or incapable of handling such systems.

This chapter, which is mostly based on two papers of the author’s [17, 20], is organized as follows. Sect. 4.2 introduces well-structured transition systems and the classical algorithm for their verification. Our approach to the problem is developed in Sect. 4.3, and Sect. 4.4 presents the

aforementioned conjunctive guard reduction. The chapter concludes with Sect. 4.5, which strives to evaluate the strengths and weaknesses of our approach.

4.2 Well-Structured Transition Systems

Here we survey the relevant concepts and previous results for well-structured transition systems, which were first proposed by Finkel [56]. Sect. 4.2.1 gives preliminary definitions. Sect. 4.2.2 presents two examples of WSTSs, namely, petri nets and broadcast protocols. Sect. 4.2.3 defines a verification problem on WSTSs and outlines the classical algorithm for its decision.

4.2.1 Definitions

A *transition system* $(S, \rightarrow)$ can be thought of as a stripped-down automaton in which the notions of symbol alphabet, accepting states, and initial state are all removed. We are left with two components: the state space S and the transition relation $\rightarrow \subseteq S \times S$. A transition system is deemed to be well-structured if there exists a certain ordering on the state space such that the transition relation is in some sense monotonic with respect to this ordering. We are interested in an ordering called a *well-quasi-ordering*, which is a special type of preorder. A *preorder*[1] is a reflexive and transitive relation. We will typically use $\preccurlyeq$ to denote a preorder and $\prec$ to denote the strict counterpart of $\preccurlyeq$, i.e. $x \prec y$ means $x \preccurlyeq y \wedge y \not\preccurlyeq x$.

Definition 38 (well-quasi-ordering). *A* well-quasi-ordering (wqo) *is a preorder $\preccurlyeq$ on a set X such that for any infinite sequence $x_0, x_1, x_2, \dots$ over X, there exists $i, j \in \mathbb{N}$ such that $i < j$ and $x_i \preccurlyeq x_j$.*

In the verification problems we will be tackling, we will require a wqo over the state space, and the set of "bad" states that we wish to check avoidance of will be characterized by sets that are *upward-closed* with respect to the wqo. Upward closure and other relevant notions are now defined, mostly following the terminology of Finkel and Schnoebelen [57]. We note that Defs. 39 and 40 assume a preorder $\preccurlyeq$ over some set X.

Definition 39 (upward-closure,upward-closed set). *For $Y \subseteq X$, the* upward-closure *of Y is the set $\uparrow Y = \{x \mid \exists y \in Y. y \preccurlyeq x\}$. A set $U \subseteq X$ is said to be* upward-closed *if $U = \uparrow U$.*

[1] also called a *quasi-ordering* in some literature

Definition 40 (basis,canonical basis). *If sets $U, Y \subseteq X$ are such that $U = \uparrow Y$ we say that Y is a* basis *of U. Further, if Y has the property that for all distinct $y_1, y_2 \in Y$ we have that y_1 and y_2 are incomparable under $\preccurlyeq$, then we say that Y is a* canonical basis *of U.*

Lemma 33 below asserts that any upward-closed set has a finite basis. Since upward-closed sets are typically infinite, this affords one a finite representation of such infinite sets.

Lemma 33 ([76]). *For a wqo, any upward-closed set U has a unique canonical basis* $\mathsf{basis}(U)$*, and this basis is finite.*

We are now equipped to define well-structured transition systems. Note that Fig. 4.1 graphically depicts the relationship between the wqo and the transition relation required by item 3 of Def. 41.

Definition 41 (well-structured transition system). *A* well-structured transition system *(WSTS) is a triple $(S, \rightarrow, \preccurlyeq)$ such that*

1. *$(S, \rightarrow)$ is a transition system*
2. *$\preccurlyeq$ is a wqo over S*
3. *For all $x, x', y \in S$ such that $x \rightarrow x'$ and $x \preccurlyeq y$, there exists $y' \in S$ such that $y \rightarrow y'$ and $x' \preccurlyeq y'$.*[2]

4.2.2 Examples

There are many types of WSTSs. In fact, by selecting the appropriate wqo, *any* transition system can be made into a WSTS [57].[3] As this thesis concentrates on shared-memory protocols, here we describe two classes of WSTSs that are perhaps the most relevant in this domain: petri nets and broadcast protocols.

[2]This requirement is called *monotonicity* by Abdulla et al. [2] and *strong compatibility* by Finkel and Schnoebelen [57]. The latter paper gives a slightly weaker definition of WSTS, requiring that y' only satisfy $y \rightarrow^* y'$.

[3]However, the verification algorithms require the bad states to be upward-closed with respect to the wqo, and as Finkel and Schnoebelen [57] point out, the wqo constructed in the proof of their *ubiquity* theorem cannot express a very interesting class of errors.

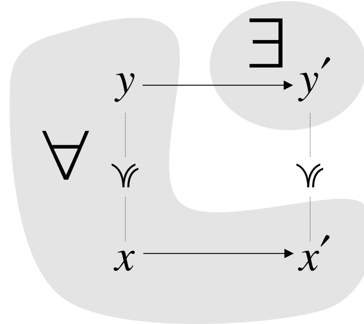


Figure 4.1: The relationship between $\preccurlyeq$ and $\rightarrow$ required by Def. 41

Petri Nets

The most widely studied class of WSTSs are petri nets,[4] which have been accused of causing the deforestation of Europe.[5] Murata [102] gives a comprehensive survey of petri net research up to 1989; Esparza [54] surveys results on complexities and computability of petri net problems. Our development follows the definitions used by Finkel and Schnoebelen [57].

Definition 42 (petri net). *A* petri net *is a triple* (P, T, F) *where* $P = \{p_1, \ldots, p_m\}$ *is a finite set of* places *which is disjoint from the finite set* T *of* transitions. $F : ((P \times T) \cup (T \times P)) \rightarrow \mathbb{N}$ *is called the* flow matrix. *A* marking *of a petri net is an element of* $\mathbb{N}^m$.

Intuitively, a marking v assigns a number of *tokens* (zero or more) to each place of the petri net; the number of tokens on place p_i is specified by $v(i)$. Transitions may *fire*, causing some tokens to move between places, be destroyed, or be created; the flow matrix defines both the guard and the resultant token activity for the firing of each transition. Before formally defining petri net semantics, we introduce the wqo on petri net markings that we will employ.

Definition 43 (inclusion ordering). *For* $m \geq 1$, *the* inclusion ordering $\preccurlyeq_{\mathsf{M}} \subseteq \mathbb{N}^m \times \mathbb{N}^m$ *is defined by* $v \preccurlyeq_{\mathsf{M}} u$ *iff for all* $i \in \mathbb{N}_m$ *we have* $v(i) \leq u(i)$.

[4]Petri nets are very closely related to the *vector addition systems* of Karp and Miller [81]; the two formalisms are of equal computational power.

[5]This joke is attributed to David Dill.

Lemma 34 (Dickson's Lemma [44]). *For any $m \geq 1$, the inclusion ordering $\preccurlyeq_{\mathsf{M}}$ over $\mathbb{N}^m$ is a wqo.*

Two markings are related by the inclusion ordering if one is greater or equal than the other component-wise. We also define addition and subtraction on markings to happen component-wise; note that the result of a subtraction operation can yield negative components, hence markings are not closed under subtraction.

To give semantics to petri nets, we first require the following notions. The *pre-vector* of transition $t \in T$ is the vector ${}^{\bullet}t$ defined by ${}^{\bullet}t(i) = F((p_i, t))$ for each $i \in \mathbb{N}_m$. Analogously, the *post-vector* $t^{\bullet}$ of t is defined by $t^{\bullet}(i) = F((t, p_i))$. For a marking v and transition t, we will say that t *may fire* from v if ${}^{\bullet}t \preccurlyeq_{\mathsf{M}} v$. If t may fire from v, and $v' = v - {}^{\bullet}t + t^{\bullet}$, we say that v' is obtained from v *by firing* t.

Definition 44 (petri net semantics). *The semantics of a petri net (P, T, F) is the transition system $(\mathbb{N}^{|P|}, \rightarrow)$, where $v \rightarrow v'$ iff there exists $t \in T$ such that v' is obtained from v by firing t.*

Lemma 35 ([2, 57]). *Let $(\mathbb{N}^m, \rightarrow)$ be the semantics of a petri net. Then $(\mathbb{N}^m, \rightarrow, \preccurlyeq_{\mathsf{M}})$ is a WSTS.*

Petri nets have a well-known graphical representation where places are circles, transitions are rectangles, and the flow matrix is indicated by arcs between places and transitions The arcs are possibly annotated with positive integers if the flow matrix has value greater than 1 for its endpoints. Figures 4.5, 4.6, and 4.7 later in this chapter are examples of this representation.

Broadcast Protocols

Broadcast protocols were first proposed by Emerson and Namjoshi [53] as a model of systems composed of an unbounded number of identical communicating finite state processes. Broadcast protocols are closely related to a generalization of petri nets called *petri nets with transfer arcs* [57, 72]. Here we roughly follow the definitions of Esparza et al. [55] and Emerson et al. [53].

Definition 45 (Broadcast protocol). *A* broadcast protocol *(BP) is a triple (L, Σ, R), where L is the set of* local states*, Σ is the set of* labels*, and $R \subseteq L \times \Sigma \times L$. Σ is required to be of the form $\Sigma_l \cup \Sigma_r \times \{!, ?\} \cup \Sigma_b \times \{!!, ??\}$, where Σ_l, Σ_r, and Σ_b are disjoint sets of primitive labels called* actions*, and are respectively called* local*,* rendezvous*, and* broadcast *actions.*

Labels of the form (a, d) are written simply as ad, i.e. $(a, ??)$ is written $a??$. Intuitively, labels of the form $a!$ and $a!!$, are outputs, while those of the form $a?$ and $a??$ are inputs; here *output* and *input* refers to inter-process communication. We make the following restriction on R: for any $a!! \in \Sigma$ and any $s \in L$, there exists $s' \in L$ such that $(s, a??, s') \in R$.

A state of our broadcast protocols is a finite word over L, the symbols of which specify the local state of each process. In the transition system defined by a broadcast protocol there are three types of transitions, corresponding to the three types of actions of Def. 45. Local transitions allow a single process to autonomously change its local state. Rendez-vous transitions model a pair-wise directional communication event between two processes. Broadcast transitions, the most powerful communication mechanism, involve a single process sending a message to *all* other processes. We now formalize these concepts.

Definition 46 (BP semantics). *The semantics of a BP (L, Σ, R) is the transition system $(L^*, \rightarrow)$ where $s \rightarrow s'$ iff $s = \ell_1 \ldots \ell_n$ and $s' = \ell'_1 \ldots \ell'_n$ for some $n \geq 1$, and one of the following hold.*

- local transition*: there exists $i \in \mathbb{N}_n$ and an action $a \in \Sigma_l$ such that $(\ell_i, a, \ell'_i) \in R$, and $\ell'_j = \ell_j$ for all $j \in \mathbb{N}_n \setminus \{i\}$.*
- rendezvous transition*: there exist distinct $i, k \in \mathbb{N}_n$ and an action $a \in \Sigma_r$ such that $(\ell_i, a!, \ell'_i) \in R$ and $(\ell_k, a?, \ell'_k) \in R$, and $\ell'_j = \ell_j$ for all $j \in \mathbb{N}_n \setminus \{i, k\}$.*
- broadcast transition*: there exists $i \in \mathbb{N}_n$ and an action $a \in \Sigma_b$ such that $(\ell_i, a!!, \ell'_i) \in R$ and, for each $j \in \mathbb{N}_n \setminus \{i\}$, we have $(\ell_j, a??, \ell'_j) \in R$.*

We note that our BP semantics differ from that of other authors [53, 55] in what is taken to constitute a state. Here we define a BP state to be an element of L^*. However, since BP processes are identical, any two words that have the same number of processes in each local state are obviously equivalent under a symmetry reduction. In other works [53, 55], this symmetry reduction is rolled into the definition of the BP semantics: a state s is a vector of $\mathbb{N}^{|L|}$, with the interpretation that the component $s(i)$ counts the number of processes in local state i. The two points of view are equivalent; we have chosen the "states-as-words" semantics simply because our algorithm is more amenable for use with this representation.[6]

[6]Indeed, a petri net can be seen as the composition of a finite number of identical processes; our development of petri nets in Sect. 4.2.2 could very well have also used the states-as-words semantics by viewing each token as a process whose local state is its place.

The wqo used for our BPs is the *subword ordering*, which is defined such that $w_1 \preccurlyeq_{\mathsf{W}} w_2$ if and only if w_1 can be obtained from w_2 by deleting 0 or more symbols. The fact that $\preccurlyeq_{\mathsf{W}}$ is a wqo is known as as *Higman's Lemma* [76]. That a BP with states in $\mathbb{N}^m$ along with $\preccurlyeq_{\mathsf{M}}$ is a WSTS was shown by Esparza et al. [55]; their result clearly applies also to our definition of BP semantics with the wqo $\preccurlyeq_{\mathsf{W}}$.

4.2.3 The Classical Algorithm

Before describing the classical algorithm, we must define the verification problem under consideration. Given a WSTS $(S, \rightarrow, \preccurlyeq)$, an upward closed set $U \subseteq S$ of bad states, and an initial state $s_0 \in S$, we wish to decide if there exists a path through $\rightarrow$ from s_0 to a state in U. We will call this decision problem the *WSTS covering problem*, and we call a path from s_0 to U an *error*. The classical algorithm for deciding this problem [2,57] exploits the following lemma about WSTS-preimages of upward-closed sets.

Lemma 36. *In a WSTS, if U is upward-closed, then so too are* $\mathsf{pre}(U) = \{x \mid \exists u \in U.x \rightarrow u\}$ *and* $\bigcup_{i=0}^{\infty} \mathsf{pre}^i(U)$.

Proof. Choose any $x \in \mathsf{pre}(U)$, and let y be any state such that $x \preccurlyeq y$. Then there exists $x' \in U$ such that $x \rightarrow x'$. From item 3 of Def. 41, there exists y' such that $y \rightarrow y'$ and $x' \preccurlyeq y'$, which implies that $y' \in U$ since U is upward-closed. Therefore $y \in \mathsf{pre}(U)$ also, and $\mathsf{pre}(U)$ is upward-closed. To see that $\bigcup_{i=0}^{\infty} \mathsf{pre}^i(U)$ is upward-closed, note that by induction, $\mathsf{pre}^i(U)$ is upward-closed for any $i \geq 0$. By Lemma 37 below, there exists some $k \geq 0$ such that for all $i \geq k$ we have $\mathsf{pre}^i(U) = \mathsf{pre}^k(U)$, thus $\bigcup_{i=0}^{\infty} \mathsf{pre}^i(U) = \bigcup_{i=0}^{k} \mathsf{pre}^i(U)$. That this set is upward-closed follows from the fact that upward-closed sets are closed under finite union. □

Also required for the decidability of the WSTS covering problem is a side-condition called the *effective pred-basis*:[7] given any finite subset $Y \subseteq S$, we must be able to compute a finite set $\mathsf{pb}(Y)$ satisfying[8]

$$\uparrow\mathsf{pb}(Y) \ = \ \mathsf{pre}(\uparrow Y) \tag{4.1}$$

[7] Abdulla et al. [2] actually roll this condition into their definition of WSTS; we follow Finkel et al. [57] and keep it separate.

[8] Note that (4.1) does not necessarily require that $\mathsf{pb}(Y) = \mathsf{basis}(\mathsf{pre}(\uparrow Y))$, only that $\mathsf{pb}(Y) \supseteq \mathsf{basis}(\mathsf{pre}(\uparrow Y))$.

1: $LastReach, Reach$: finite subset of S
2: $LastReach := \emptyset$
3: $Reach := \mathsf{basis}(U)$
4: **while** $\uparrow Reach \not\subseteq \uparrow LastReach$ **do**
5: **if** $s_0 \in \uparrow Reach$ **then**
6: exit "error found"
7: **end if**
8: $LastReach := Reach$
9: $Reach := Reach \cup \mathsf{pb}(Reach)$
10: **end while**
11: exit "verification successful"

Figure 4.2: The classical algorithm for the WSTS covering problem

That the RHS of (4.1) is upward-closed follows from Lemma 36. In the case that such a function pb is computable, the WSTS is said to have an *effective pred-basis*. All WSTSs we consider in this chapter have effective pred-basis, therefore this assumption is implicit from now on.

The classical algorithm for the WSTS covering problem [2, 55, 57] is given in Fig. 4.2. On the surface, this algorithm resembles the well-known finite-state backward reachability analysis, i.e. *least fix-point computation* [24], the difference being that the involved sets are upward-closed (and hence not necessarily finite), so a symbolic representation via a finite basis is necessary. Note that the conditional expressions of lines 4 and 5 are both clearly decidable; for line 4 one simply checks that for all $x \in Reach$ there exists $y \in LastReach$ such that $y \preccurlyeq x$; if so then the containment holds and the loop terminates. Similarly, for line 5 one checks if there exists $x \in Reach$ such that $x \preccurlyeq s_0$.

Analogously to finite-state reachability analysis, the algorithm of Fig. 4.2 has the following invariant: after the ith iteration of the while loop, $Reach$ is a finite basis for the set of states that can reach U in at most i steps. It is easy to see that the algorithm is correct *if it terminates*. Furthermore, if there exists an error, clearly line 6 will eventually be executed, terminating the algorithm. That termination is always guaranteed in the absence of errors follows from Lemma 36 and the following Lemma 37.

Lemma 37 ([57]). *With respect to a wqo, for any infinite sequence of upward-closed sets* $U_0 \subseteq U_1 \subseteq U_2 \subseteq \cdots$, *there exists* i *such that* $U_i = U_j$ *for all* $j \geq i$.

Since $Reach$ grows monotonically at each iteration of the while-loop of Fig. 4.2, so too does $\uparrow Reach$ grow monotonically. Lemma 37 thus guarantees that we will eventually have

$\uparrow Reach = \uparrow LastReach$, causing the while-loop to terminate.

Necessary for practical implementation of the classical algorithm is an efficient representation of $Reach$, since this set can become very large. Delzanno et al. propose using a data structure called *covering sharing trees* (CSTs) for this purpose [42, 43]. CSTs are applicable when the state space is $\mathbb{N}^m$; they represent a finite subset of $\mathbb{N}^m$ similarly to how a BDD [23] represents a subset of $\mathbb{B}^m$. We will discuss CSTs and the classical algorithm further in Sect. 4.5.2.

4.3 Our Approach

In this section we present our approach to deciding the WSTS covering problem. We handle a subclass of WSTSs we call *nicely sliceable WSTSs* (NSWs), which are defined in Sect. 4.3.1. Sect. 4.3.2 develops our algorithm that solves the WSTS covering problem for NSWs, along with its supporting Convergence Theorem and Termination Theorem. Sects. 4.3.3 and 4.3.4 improve our algorithm by respectively accommodating certain infinite sets of initial states and performing an important optimization. Finally, Sect. 4.3.5 shows that the Convergence Theorem is near-optimal, however determining exact optimality is left as an open problem.

4.3.1 Nicely Sliceable WSTS

To be deemed a NSW, a WSTS must satisfy three properties. We first describe each intuitively and provide some motivation for why they are required, and then we present the formal definitions.

- **Discrete:** The wqo must be discrete, meaning that for any element x, there is a bound on the length of any strictly decreasing sequence starting with x. We call the length of the longest such sequence x's *weight*. Furthermore, discreteness requires that the number of elements of a given weight be finite. Discreteness allows for finite-state model checking to be applied to the subsystem formed by bounding the weight of states.

- **Weight-respecting:** When a transition changes the weight, the same change in weight can be effected by the transition relation for elements greater than (in the wqo) the starting state of the transition. Weight-respectfulness is a technical requirement needed for the proof of our Convergence Theorem, which gives a termination condition for our algorithm.

- **Deflatable:** Whenever we have a transition from outside an upward-closed set U to a state in U, deflatability asserts the existence of a similar transition involving states of bounded weight. Deflatability is similar to downward compatibility [57], though the two are incomparable. Deflatability, like weight-respectfulness, is essential in the proof of our Convergence Theorem.

We now formalize these properties, using petri nets as a running example.

Definition 47 (dwqo,weight function,base weight,discrete WSTS). *A wqo is a* discrete wqo *(dwqo) over X if for all $x \in X$ there exists $k \in \mathbb{N}$ such that for any sequence $x_0 \prec x_1 \prec \cdots \prec x_\ell = x$ we have $\ell \leq k$. Associated with a dwqo $\preccurlyeq$ is the* weight function $\mathsf{w} : X \to \mathbb{N}$ *that maps each x to the minimum such k. We also require that $\{x \in X \mid \mathsf{w}(x) = i\}$ be finite for each $i \in \mathbb{N}$. For upward-closed U, the* base weight *of U is* $\mathsf{bw}(U) = \max(\{\mathsf{w}(x) \mid x \in \mathsf{basis}(U)\})$. *A* discrete WSTS *(DWSTS) is a WSTS $(S, \to, \preccurlyeq)$ where $\preccurlyeq$ is a dwqo.*

The inclusion ordering $\preccurlyeq_{\mathsf{M}}$ over $\mathbb{N}^m$ is a dwqo, and it is easy to show that the induced weight function is such that $\mathsf{w}(v) = \sum_{i=1}^{m} v(i)$. Thus petri nets are DWSTS, and the weight of a marking is simply the number of tokens. The set $\{0, 1/2, 2/3, 3/4, \ldots\} \cup \{1\}$ along with $\leq$ is an example of a wqo that *is not* a dwqo, since taking $x = 1$ violates Def. 47. If X is a dwqo ordered set and $i \geq 0$ is an integer, we denote by $X_{\leq i}$ the set $\{x \in X \mid \mathsf{w}(x) \leq i\}$; note that it follows from Def. 47 that $X_{\leq i}$ is always finite.

Definition 48 (weight respecting DWSTS). *A DWSTS is said to be* weight respecting *if we may strengthen condition 3 of Def. 41 to require that* $\mathsf{w}(x') - \mathsf{w}(x) = \mathsf{w}(y') - \mathsf{w}(y)$.

Petri nets are weight respecting DWSTSs. Suppose $x \to x'$ by firing transition t, and y is such that $x \preccurlyeq_{\mathsf{M}} y$. Then ${}^\bullet t \preccurlyeq_{\mathsf{M}} x$ and $x' = x - {}^\bullet t + t^\bullet$, which implies ${}^\bullet t \preccurlyeq_{\mathsf{M}} y$ by the transitivity of $\preccurlyeq_{\mathsf{M}}$. Let $y' = x' + (y - x)$, then clearly $y' \in \mathbb{N}^m$ (i.e. y' has no negative components) since $y - x \in \mathbb{N}^m$, and further $y' = x - {}^\bullet t + t^\bullet + (y - x) = y - {}^\bullet t + t^\bullet$. Thus y' is obtained from y by firing t, and $y \to y'$.

Definition 49 (δ-deflatable DWSTS). *A DWSTS $(S, \to, \preccurlyeq)$ is said to be* δ-deflatable *for $\delta \in \mathbb{N}$ if whenever $x \to x'$ and $z \preccurlyeq x'$, there exist y and y' such that all of the following hold:*

1. $y \preccurlyeq x$

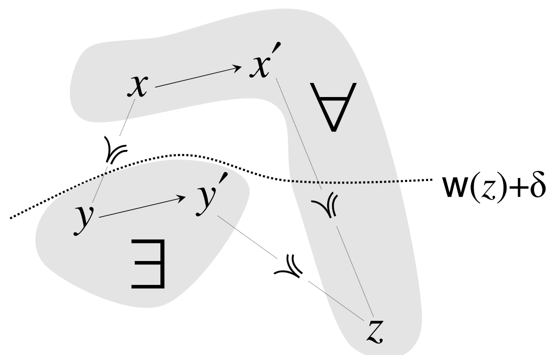


Figure 4.3: A diagrammatic presentation of Def. 49. A DWSTS $(S, \rightarrow, \preccurlyeq)$ is said to be δ-deflatable (for $\delta \in \mathbb{N}$) if for all $x, x', z \in S$ that satisfy the depicted relations, there exist $y, y' \in S$ that satisfy the depicted relations, and also both $\mathsf{w}(y)$ and $\mathsf{w}(y')$ are not greater than $\mathsf{w}(z) + \delta$.

2. $y \rightarrow y'$
3. $z \preccurlyeq y'$
4. $\mathsf{w}(y) \leq \mathsf{w}(z) + \delta$
5. $\mathsf{w}(y') \leq \mathsf{w}(z) + \delta$

Fig. 4.3 shows the requirements of a δ-deflatable DWSTS. z should be thought of as an element in some upward-closed set with relatively low weight. Viewed contrapositively, δ-deflatability asserts that if there is no transition $y \rightarrow y'$ into a state y' that is above z in the dwqo such that the weights of y and y' do not exceed that of z by more than δ, then there is no transition (involving states of *any* weight) into a state above z. Intuitively, this implies that upward-closed U can be shown to be a fixpoint of pre (meaning $U = \mathsf{pre}(U)$) by only considering states of weight at most $\mathsf{bw}(U) + \delta$.

Our running example of petri nets are deflatable. The proof of this is sufficiently involved to warrant its own lemma.

Lemma 38. *Under $\preccurlyeq_{\mathsf{M}}$, a petri net (P, T, F) is δ-deflatable, where*

$$\delta = \max\left(\{\mathsf{w}({}^{\bullet}t) \mid t \in T\} \cup \{\mathsf{w}(t^{\bullet}) \mid t \in T\}\right)$$

Proof. Suppose we have x, x', and z as in Def. 49, and let t be the transition that fires to take x to x'. For the purposes of this proof, we introduce the *monus* operator $\ominus : \mathbb{N}^{|P|} \times \mathbb{N}^{|P|} \to \mathbb{N}^{|P|}$ where $v \ominus u$ is the vector w such that

$$w(i) = \begin{cases} v(i) - u(i) & \text{if } v(i) \geq u(i) \\ 0 & \text{otherwise} \end{cases}$$

Let us define $y = (z \ominus t^\bullet) + {}^\bullet t$ and $y' = (z \ominus t^\bullet) + t^\bullet$. It is easy to see that conditions 2-5 of Def. 49 hold of these y and y'; the remainder of the proof is devoted to showing condition 1. Since we can fire t from x to get x', we have that ${}^\bullet t \preccurlyeq_{\mathsf{M}} x$ and $x' = x - {}^\bullet t + t^\bullet \succcurlyeq_{\mathsf{M}} z$. Thus

$$x - {}^\bullet t \succcurlyeq_{\mathsf{M}} z - t^\bullet \tag{4.2}$$

Now since the LHS of (4.2) has no negative components, we may safely replace the $-$ in the RHS with $\ominus$, since this replacement simply increases all negative components to 0. Hence $x - {}^\bullet t \succcurlyeq_{\mathsf{M}} z \ominus t^\bullet$, which implies $x \succcurlyeq_{\mathsf{M}} (z \ominus t^\bullet) + {}^\bullet t = y$. $\square$

Definition 50 (NSW). *A* δ-NSW *is a DWSTS that is weight-respecting and* δ*-deflatable. A* NSW *is a* δ*-NSW for some* δ*.*

From our earlier discussion and Lemma 38 we find that petri nets are δ-NSWs, where δ is as in the statement of Lemma 38. In a previous paper [20] we argued that two other classes of WSTSs are NSWs, namely context-free grammars and lossy channel systems. Since this thesis is about shared-memory protocols, here we show that broadcast protocols are also NSWs.

Theorem 14. *Broadcast protocols along with the subword ordering are* 2*-NSWs.*

Proof. Clearly the sub-word ordering $\preccurlyeq_{\mathsf{W}}$ is discrete and the weight of a word is simply its length. Noting that all BP transition types of Def. 45 preserve the length of the word, we see that BPs are trivially weight-respecting. We now show that any BP (L, Σ, R) is 2-deflatable. Suppose we have words x, x', and z as in Def. 49. Let $x = x_1 \ldots x_n$ and $x' = x'_1 \ldots x'_n$; we will automatically expand the words y, y', and z similarly. For $D = \{d_1, d_2, \ldots, d_e\} \subseteq \mathbb{N}_n$ where $d_1 < d_2 < \cdots < d_e$, let x_D be the word $x_{d_1} x_{d_2} \ldots x_{d_e}$; we apply the same notation to the other words also. Then we have $z = x'_D$ for some $D \subseteq \mathbb{N}_n$, and $\mathsf{w}(z) = |D|$. Also, for $X \subseteq \mathbb{N}_n$ and $i \in X$, let $r(X, i) = |X \cap \mathbb{N}_i|$. The importance of r in this proof is summarized by the fact that for any word w, $X \subseteq \mathbb{N}_{|w|}$, and $i \in X$ we have $w_i = u_{r(X,i)}$, where $u = w_X$.

We case-split on the type of the transition $x \to x'$. In all cases conditions 1 and 3 of Def. 49 will obviously hold of the y and y' we select, hence we focus on conditions 2, 4, and 5. Note that local and broadcast cases are very similar; the rendezvous case is slightly different.

- *local transition*. Let i and a be as in the local transition case of Def. 46. Then let $y = x_{D \cup \{i\}}$, let $y' = x'_{D \cup \{i\}}$, and let $\bar{i} = r(D \cup \{i\}, i)$. It follows that $(y_{\bar{i}}, a, y'_{\bar{i}}) \in R$ and $y_j = y'_j$ for all $j \in \mathbb{N}_{|y|} \setminus \{\bar{i}\}$, and thus $y \to y'$ is a local transition. Since $\mathsf{w}(y') = \mathsf{w}(y) = |D \cup \{i\}| \leq \mathsf{w}(z) + 1$, conditions 4 and 5 of Def. 49 hold.

- *rendezvous*. Let i, k, and a be as in the rendezvous transition case of Def. 46. Then let $y = x_{D \cup \{i,k\}}$, let $y' = x'_{D \cup \{i,k\}}$, let $\bar{i} = r(D \cup \{i,k\}, i)$ and let $\bar{k} = r(D \cup \{i,k\}, k)$. It follows that $(y_{\bar{i}}, a!, y'_{\bar{i}}) \in R$ and $(y_{\bar{k}}, a?, y'_{\bar{k}}) \in R$ and $y_j = y'_j$ for all $j \in \mathbb{N}_{|y|} \setminus \{\bar{i}, \bar{k}\}$ and thus $y \to y'$ is a rendezvous transition. Since $\mathsf{w}(y') = \mathsf{w}(y) = |D \cup \{i,k\}| \leq \mathsf{w}(z) + 2$, conditions 4 and 5 of Def. 49 hold.

- *broadcast*. Let i and a be as in the broadcast transition case of Def. 46. Then let $y = x_{D \cup \{i\}}$, let $y' = x'_{D \cup \{i\}}$, and let $\bar{i} = r(D \cup \{i\}, i)$. It follows that $(y_{\bar{i}}, a!!, y'_{\bar{i}}) \in R$ and $(y_j, a??, y'_j) \in R$ for all $j \in \mathbb{N}_{|y|} \setminus \{\bar{i}\}$, and thus $y \to y'$ is a broadcast transition. Since $\mathsf{w}(y') = \mathsf{w}(y) = |D \cup \{i\}| \leq \mathsf{w}(z) + 1$, conditions 4 and 5 of Def. 49 hold.

In each case we have $\mathsf{w}(y'), \mathsf{w}(y) \leq \mathsf{w}(z) + 2$. □

4.3.2 Our Algorithm

This section develops our algorithm for the WSTS covering problem, which is shown in Fig. 4.4. The inputs are the same as for the classical algorithm of Sect. 4.2.3: a WSTS $(S, \to, \preccurlyeq)$, an initial state $s_0 \in S$, and an upward-closed (w.r.t $\preccurlyeq$) set of bad states U. We assume that the WSTS is in fact a δ-NSW for some δ; to emphasize this assumption we will call the problem we solve the *NSW covering problem*. Note that the actual value of δ will depend on the NSW description according to some predetermined rule (such as our Theorem 14).

For each $i = \mathsf{bw}(U), \mathsf{bw}(U){+}1, \mathsf{bw}(U){+}2, \ldots$, our algorithm computes the backward reachable set $\mathsf{br}(U, i)$, which is the set of states from which U is reachable along a path that never exceeds weight i, and is defined now.

Definition 51 (br). *Given a DWSTS $(S, \rightarrow, \preccurlyeq)$, a set $Y \subseteq S$, and $i \in \mathbb{N} \cup \{\infty\}$ we let* $\mathsf{br}(Y, i)$ *denote the set of all $x \in S$ such that there exists a sequence $x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_\ell$ such that $x_0 = x$, $x_\ell \in Y$, and for all $0 \leq j \leq \ell$ we have $\mathsf{w}(x_j) \leq i$. We also define $\mathsf{br}(Y) = \mathsf{br}(Y, \infty)$.*[9]

Since $\mathsf{br}(U, i)$ is necessarily finite for all $i \geq 0$, this set can be computed using classical finite-state symbolic model checking [24] based on BDDs [23]. Our algorithm computes $\mathsf{br}(U, i)$ for increasingly large values of i, and terminates upon either of the following events:

- Convergence occurs. By *convergence*, we mean that we have reached an i such that $\uparrow \mathsf{br}(U, i) = \mathsf{br}(U)$. How we decide if this equality holds for a given i is articulated in Theorem 15 below. The existence of such an i is guaranteed by Theorem 16.
- An error is detected, i.e. we reach an i such that $s_0 \in \mathsf{br}(U, i)$.

We now present two theorems. Theorem 15 gives us a necessary and sufficient condition for convergence, while Theorem 16 guarantees that our algorithm will always terminate.

Theorem 15 (Convergence). *For a δ-NSW, an upward-closed set U, and $n \geq \mathsf{bw}(U)$,*

$$\mathsf{br}(U, n + \delta) \subseteq \uparrow\mathsf{br}(U, n) \tag{4.3}$$

if and only if

$$\mathsf{br}(U) = \uparrow\mathsf{br}(U, n) \tag{4.4}$$

Proof. ($\Leftarrow$) Trivial, since $\mathsf{br}(U, n + \delta) \subseteq \mathsf{br}(U)$. ($\Rightarrow$) $\mathsf{br}(U) \supseteq \uparrow\mathsf{br}(U, n)$ follows from Lemma 36 and the fact that $\mathsf{br}(U) \supseteq \mathsf{br}(U, n)$. To prove the converse containment, suppose that (4.3) holds, but there exist $i \geq 1$ and $x \in \mathsf{br}(U, i)$ such that $x \notin \uparrow\mathsf{br}(U, n)$. Since $\mathsf{br}(U, j) \subseteq \mathsf{br}(U, k)$ whenever $j \leq k$, $i \leq n + \delta$ implies $x \in \mathsf{br}(U, n + \delta) \subseteq \uparrow\mathsf{br}(U, n)$, thus we need only consider $i > n + \delta$.

Let $x = x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_\ell \in U$ be a path from x to U. Since $U \subseteq \uparrow\mathsf{br}(U, n)$, there must exist $k \in \{1, \ldots, \ell\}$ such that $x_k \in \uparrow\mathsf{br}(U, n)$ and $x_{k-1} \notin \uparrow\mathsf{br}(U, n)$. Then there exists $z \in \mathsf{br}(U, n)$ such that $z \preccurlyeq x_k$. Now because the system is δ-deflatable, there exist y and y' satisfying the five conditions of Def. 49, i.e. $y \preccurlyeq x_{k-1}$, $y \rightarrow y'$, $z \preccurlyeq y'$, $\mathsf{w}(y) \leq \mathsf{w}(z) + \delta$, and $\mathsf{w}(y') \leq \mathsf{w}(z) + \delta$. From Lemma 39 below, we have that $y' \in \mathsf{br}(U, n + \delta)$ which implies $y \in \mathsf{br}(U, n + \delta)$ since $\mathsf{w}(y) \leq n + \delta$. Since $y \preccurlyeq x_{k-1}$, this implies $x_{k-1} \in \uparrow\mathsf{br}(U, n)$, which is a contradiction. ∎

[9] Note that $\mathsf{br}(Y)$ is also equal to $\bigcup_{i=0}^{\infty} \mathsf{pre}^i(Y)$.

Lemma 39. *For a weight respecting DWSTS and an upward-closed set U, $z \in \mathsf{br}(U, n)$ and $z \preccurlyeq y'$ imply $y' \in \mathsf{br}(U, n + \mathsf{w}(y') - \mathsf{w}(z))$.*

Proof. Let $d = \mathsf{w}(y') - \mathsf{w}(z)$. $z \in \mathsf{br}(U, n)$ implies there exists a sequence $z_0 \to z_1 \to \cdots \to z_\ell$ where $z_0 = z$ and $z_\ell \in U$, and $\mathsf{w}(z_j) \leq n$ for all $0 \leq j \leq \ell$. We show by induction on ℓ that $y' \in \mathsf{br}(U, n + d)$. If $\ell = 0$, then $z \in U$ and thus $y' \in U$ and we are done since $\mathsf{w}(y') \leq n + d$. For the inductive step, we note that $z \to z_1$ implies that there exists y_1 such that $\mathsf{w}(y_1) = \mathsf{w}(z_1) + d$, $y' \to y_1$, and $z_1 \preccurlyeq y_1$ since the system is weight respecting. Note that $z_1 \in \mathsf{br}(U, n)$, thus, by our inductive hypothesis, $y_1 \in \mathsf{br}(U, n + \mathsf{w}(y_1) - \mathsf{w}(z_1)) = \mathsf{br}(U, n + d)$, which implies $y' \in \mathsf{br}(U, n + d)$. □

One can show that (4.3) is equivalent to having

$$\mathsf{br}(U, i) \subseteq \uparrow\mathsf{br}(U, i - 1) \tag{4.5}$$

for all $i \in \{n + 1, \ldots, n + \delta\}$. We will refer to values of i for which (4.5) holds as being *stable*. Hence Theorem 15 states that if we find δ consecutive stable values, then we have convergence. The next theorem guarantees that this will eventually happen.

Theorem 16 (Termination). *For any DWSTS and upward-closed set U, there exists an n satisfying (4.4).*

Proof. For any $x \in \mathsf{br}(U)$, let $g(x)$ denote the minimum j such that $x \in \mathsf{br}(U, j)$. Since by Lemma 36 $\mathsf{br}(U)$ is upward-closed, there exists a finite set $B = \mathsf{basis}(\mathsf{br}(U))$. Now take $n = \max(\{g(x) \mid x \in B\})$. To see that this n satisfies (4.4), note that for any $i \geq n$, if there exists $x \in \mathsf{br}(U, i)$ such that $x \notin \uparrow\mathsf{br}(U, n)$, then this contradicts B being a basis for $\mathsf{br}(U)$, since $B \subseteq \mathsf{br}(U, n)$. □

In order to use Theorem 15 in our algorithm, we must have a means to decide (4.3). Our approach requires the use of a computable *lifting operator*, which intuitively "lifts" a set $\mathsf{br}(U, i)$ to a truncated version of its upward-closure. The truncation omits everything with weight strictly greater than some given $d \in \mathbb{N}$; hence finiteness is preserved.

Definition 52 (lifting operator). *Given a dwqo $\preccurlyeq$ over a set X, the associated* lifting operator *is the function* $\mathsf{Lift} : X \times \mathbb{N} \to 2^X$ *defined by*

$$\mathsf{Lift}(x, d) = \{y \mid x \preccurlyeq y \wedge \mathsf{w}(y) \leq d\}$$

1: $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$: finite subset of S
2: i, n : integer
3: $i := \mathsf{bw}(U)$
4: $n := i$
5: $\Gamma_{i-1} := \emptyset$
6: **while** $(n \geq i - \delta)$ **do**
7: $\quad\Gamma_i := \mathsf{br}(U, i)$
8: $\quad$**if** $s_0 \in \uparrow\Gamma_i$ **then**
9: $\quad\quad$exit "error found"
10: $\quad$**end if**
11: $\quad$**if** $(\Gamma_i \not\subseteq \mathsf{Lift}(\Gamma_{i-1}, i))$ **then**
12: $\quad\quad n := i$
13: $\quad$**end if**
14: $\quad i := i + 1$
15: **end while**
16: exit "verification successful"

Figure 4.4: Our algorithm for the NSW covering problem. Its inputs are a δ-NSW $(S, \rightarrow, \preccurlyeq)$, a basis of an upward-closed set U, and an initial state s_0.

We extend Lift *to act on sets by decreeing* $\mathsf{Lift}(Y, d) = \bigcup_{y \in Y} \mathsf{Lift}(y, d)$.

The following theorem explains how the lifting operator is relevant to deciding containments along the lines of (4.3) or (4.5). For a finite set X, let $\mathsf{maxw}(X) = \max(\{\mathsf{w}(x) \mid x \in X\})$

Theorem 17. *Let $\preccurlyeq$ be a dwqo over a set X, and let X_{i-1} and X_i be finite subsets of X such that* $\mathsf{maxw}(X_{i-1}) \leq i - 1$ *and* $\mathsf{maxw}(X_i) \leq i$. *Then $X_i \subseteq \uparrow X_{i-1}$ if and only if $X_i \subseteq \mathsf{Lift}(X_{i-1}, i)$.*

Proof. ($\Leftarrow$) Trivial, since $\mathsf{Lift}(X_{i-1}, i) \subseteq \uparrow X_{i-1}$. ($\Rightarrow$) Suppose $X_i \subseteq \uparrow X_{i-1}$, and let $x \in X_i$. Then there exists $y \in X_{i-1}$ such that $y \preccurlyeq x$. Since $\mathsf{w}(x) \leq i$, this implies $x \in \mathsf{Lift}(X_{i-1}, i)$. $\square$

Our algorithm for the NSW covering problem is given in Fig. 4.4. As inputs, the algorithm takes a δ-NSW, a basis of an upward-closed set U, and an initial state s_0. The variable i represents the maximum weight of the states computed in each iteration of the while-loop. i is initially the base weight of U and is incremented each iteration. The variable n tracks the last value of i for which "new" states were found in $\mathsf{br}(U, i)$, i.e. states x that weren't already "covered" by the existence of $y \in \mathsf{br}(U, i-1)$ such that $y \preccurlyeq x$. The condition of the while loop (line 6) will only fail when (4.3) holds, which by Theorem 15 indicates convergence. Each iteration of the loop involves computing $\mathsf{br}(U, i)$, which is done in a nested preimage fix-point computation (implicit in line 7). Line 8 tests

to see if the initial states have been reached, and line 9 terminates if so. Line 11 determines if something "new" was found this iteration, if so n is updated to be i. If the condition of line 11 fails δ times consecutively, by Theorem 17 we have all of

$$\begin{array}{rcl} \mathsf{br}(U, n+\delta) & \subseteq & \uparrow\mathsf{br}(U, n+\delta-1) \\ \mathsf{br}(U, n+\delta-1) & \subseteq & \uparrow\mathsf{br}(U, n+\delta-2) \\ & \vdots & \\ \mathsf{br}(U, n+1) & \subseteq & \uparrow\mathsf{br}(U, n) \end{array}$$

(i.e. we are stable for all $i \in \{n+1, \ldots, n+\delta\}$) which implies

$$\mathsf{br}(U, n+\delta) \subseteq \uparrow\mathsf{br}(U, n)$$

Thus, by Theorem 15, (4.3) holds and verification is successful. Theorem 16 guarantees that this will eventually happen.

We now give an example of our algorithm running on the petri net (P, T, F), where $P = \{p_1, p_2, p_3\}$, $T = \{t_1, t_2, t_3\}$, $F(t_1, p_1) = F(p_1, t_2) = F(p_2, t_2) = F(t_2, p_3) = F(p_3, t_3) = F(t_3, p_2) = F(t_3, p_1) = 1$, and all other entries of F are 0. Fig. 4.5 shows this petri net using the usual graphical representation. We will write markings by juxtaposing the token counts at p_1, p_2, and p_3, for example 123 is the marking with j tokens at p_j, for each $1 \leq j \leq 3$. We desire to check if a marking placing 3 or more tokens at p_3 is reachable from the initial state $s_0 = 720$. The set of bad markings is thus the upward-closed set $U = \uparrow\{003\}$. Note that from Lemma 38 we have $\delta = 2$. Table 4.1 summarizes the execution of our algorithm, given this petri net, s_0, and U. The rows correspond to the state of the program each time line 6 is visited. The final column gives the values computed for $\Gamma_2, \Gamma_3, \ldots$. We see that convergence occurs after 4 iterations of the *while*-loop, and since $s_0 = 720$ is not in the upward-closure of any of the computed Γ_j's, the verification is successful.

The next subsection explains how our algorithm can be generalized to handle a certain class of infinite initial state sets.

4.3.3 Parametric Initial States

In Sect. 4.2.3 we defined the WSTS covering problem to check if the bad states are reachable from a *single* initial state. For systems such as petri nets and BPs, we typically desire to perform this check

VL6	n	i	Γ_{i-1}
1	3	3	$\Gamma_2 = \emptyset$
2	3	4	$\Gamma_3 = \{003\}$
3	4	5	$\Gamma_4 = \mathsf{Lift}(\Gamma_3, 4) \cup \{030, 130, 040, 031\}$
4	4	6	$\Gamma_5 = \mathsf{Lift}(\Gamma_4, 5)$
5	4	7	$\Gamma_6 = \mathsf{Lift}(\Gamma_5, 6)$

Table 4.1: Summary of the execution of our algorithm (Fig. 4.4) against example petri net of Fig. 4.5. VL6 stands for "visit to line 6"; the rows give the values of the variables n, i, and Γ_{i-1} for the first, second, etc. visit to line 6.

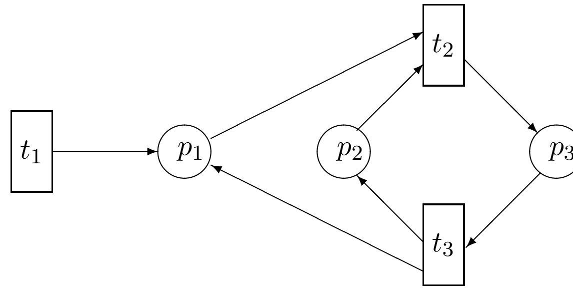


Figure 4.5: Example petri net

with respect to an *infinite set* of initial states. In fact, BP verification relative to a single initial state reduces to finite-state model checking, since all BP transitions preserve weight. In other words, BPs are only interesting when dealing with infinite initial state sets.

Both the classical algorithm of 4.2.3 and our algorithm of Sect. 4.3.2 can easily be adapted to handle an infinite set of initial states called a *parametric set* [55];[10] this section explains this adaptation for our algorithm. Here we define parametric sets of words, which can be used to express typical initial state configurations for broadcast protocols.[11]

Definition 53 (parametric set). *Let $L = \{\ell_1, \ldots, \ell_m\}$ be a finite alphabet. A subset I of L^* is said to be a* parametric set *if it can be written as*

$$I = \{w \mid \#(w, \ell_1) \sim_1 a_1 \wedge \cdots \wedge \#(w, \ell_m) \sim_m a_m\} \tag{4.6}$$

where $\#(w, \ell_j)$ is the number of occurrences of ℓ_j in w, each $\sim_j$ is either $=$ or $\geq$, and each $a_j \in \mathbb{N}$.

[10] Our notion of parametric set is slightly more general than the *parameterized configuration* of Esparza et al. [55].

[11] One can also define parametric sets of petri net markings, and our algorithm can be used for these too. A parametric set of markings specifies for each place p a constraint of the form $v(p) \sim_p n_p$, where $\sim_p$ is either $\geq$ or $=$, $n_p \in \mathbb{N}$, and $v(p)$ is the number of tokens p may have in an initial state.

For example, consider the MSI caching protocol. This is a simple BP with three local states M, S, and I. If the initial state is required to have all processes in the invalid state I, we can express this as the parametric set

$$\{w \mid \#(w, \mathsf{M}) = 0 \wedge \#(w, \mathsf{S}) = 0 \wedge \#(w, \mathsf{I}) \geq 1\}$$

Similar to an upward-closed set, a parametric set has a finite representation via the $\sim_j$'s and a_j's of (4.6). If the NSW's state space is a set of words, and the wqo is the sub-word ordering, our algorithm can be adapted to check if a parametric set of initial states is backwards reachable from the upward-closed bad states. The following Theorem 18 shows how this is done, and states that a parametric set I intersects an update-closed set U if and only if U contains a word x that doesn't include too many occurrences of the ℓ_i's involved in equality constraints in (4.6). This is because since U is upward-closed, if the number of ℓ_i's is too few, occurrences of ℓ_i can be added to x as necessary while remaining in U. Furthermore the theorem states that only the finite number of x with weight at most $\mathsf{bw}(Y)$ need be considered.

Theorem 18. *Let $I, Y \subseteq L^*$ be such that I is a parametric set of the form (4.6) and Y is upward-closed, and let $r \geq \mathsf{bw}(Y)$. Then $I \cap Y = \emptyset$ if and only if $\{x \in Y \mid \mathsf{w}(x) \leq r\}$ does not contain x such that $\#(x, \ell_j) \leq a_j$ for all j such that $\sim_j$ is $=$.*

Proof. Let $E = \{j \in \mathbb{N}_m \mid \ \sim_j \text{ is } =\}$ and let $m = |L|$.

($\Rightarrow$) Suppose $Y_{\leq r}$ contains some x as in the statement of the theorem. We define a function $f : L \to \mathbb{N}$ by

$$f(\ell_j) = \begin{cases} a_j - \#(x, \ell_j) & \text{if } \#(x, \ell_j) \leq a_j \\ 0 & \text{otherwise} \end{cases}$$

Now let $x' = x s_1 s_2 \ldots s_m$, where s_j is the string consisting of $f(\ell_j)$ copies of ℓ_j. Then clearly $x \preccurlyeq x'$, thus $x' \in Y$. Now since $\#(x, \ell_j) \leq a_j$ for all $j \in E$, we find that $\#(x', \ell_j) = a_j$ for all $j \in E$. Furthermore, $\#(x', \ell_j) \geq a_j$ for all $j \in \mathbb{N}_m \setminus E$. Therefore $x' \in I$, and $I \cap Y$ is nonempty.

($\Leftarrow$) Now suppose there exists $x \in I \cap Y$. If $\mathsf{w}(x) \leq r$ we are done, since then $\#(x, \ell_j) = a_j$ for all $j \in E$ and $x \in Y_{\leq r}$. Otherwise, there exists $x' \in \mathsf{basis}(Y)$ such that $x' \preccurlyeq x$, which implies $\#(x', \ell_j) \leq \#(x, \ell_j) = a_j$ for all $j \in E$. But since $\mathsf{w}(x') \leq \mathsf{bw}(Y) \leq r$, we have $x' \in Y_{\leq r}$. ☐

Let us assume we are given initial states of the form (4.6), and let E be as in the proof of

Theorem 18. To use Theorem 18 in our algorithm, we simply modify the conditional of the if-statement of line 8 of our algorithm to read:

$$\exists x \in \Gamma_i.\forall j \in E.\#(x, \ell_j) \leq a_j \tag{4.7}$$

That (4.7) is decidable follows from the fact that Γ_i is finite. Correctness of this modification follows from the facts that $\Gamma_i \subseteq \mathsf{br}(U)_{\leq i}$ is an invariant of our algorithm, and furthermore, assuming no error is found, Theorems 15 and 16 ensure that we will eventually arrive at a value of i for which $\mathsf{br}(U) = \uparrow\Gamma_i$.

4.3.4 An Optimization

In this section we propose an optimization to the algorithm of Fig. 4.4. Note that in Fig 4.4, the computation of $\mathsf{br}(U, i)$ of line 7 involves an iterative fix-point computation, starting with the set $U_{\leq i}$. In some sense, much of the work of this computation was already performed when computing $\mathsf{br}(U, i-1)$; since this is a subset of $\mathsf{br}(U, i)$, it is redundant to "rediscover" these states. Furthermore, many of the states with weight i in $\mathsf{br}(U, i)$, though not in $\mathsf{br}(U, i-1)$, *are* in the upward closure of $\mathsf{br}(U, i-1)$. Exerting computational effort to find such states is also redundant, since we know that $\mathsf{br}(U)$ is upward-closed (Lemma 36).

Our optimization serves to eliminate this computational redundancy, and involves replacing lines 5 and 7 of Fig. 4.4 with the following, respectively:

$$\begin{aligned} 5' : \quad \Gamma_{i-1} &:= \mathsf{basis}(U) \\ 7' : \quad \Gamma_i &:= \mathsf{br}(\mathsf{Lift}(\Gamma_{i-1}, i), i) \end{aligned} \tag{4.8}$$

Note that the fix-point now starts with $\mathsf{Lift}(\Gamma_{i-1}, i)$; however this does not incur any additional computational costs as this set must be computed anyway for the containment check of line 11. This optimization has the potential to greatly reduce the number of iterations performed in the fix-point computations. An extreme example are the stable values of i, for which the computation of Γ_i will always involve only a *single* preimage computation. In contrast, for the same value of i, the computation of Γ_i in the unoptimized algorithm will generally require many preimage iterations.

Theorem 19. *The optimization (4.8) preserves correctness of our algorithm.*

Proof. Let $b = \mathsf{bw}(U)$, and let $\Gamma_{b-1}, \Gamma_b, \Gamma_{b+1}, \ldots$ and $\Gamma'_{b-1}, \Gamma'_b, \Gamma'_{b+1}, \ldots$ be the sequences of values assigned to the variables Γ_i by the algorithm of Fig. 4.4 and the optimized version, respectively. We first show that for all $i \geq b$ we have

$$\Gamma_i \;\subseteq\; \Gamma'_i \;\subseteq\; \mathsf{br}(U) \tag{4.9}$$

(4.9) clearly holds when $i = b$, since $\Gamma_b = \Gamma'_b$. Assume that (4.9) holds for $i \geq b$. $\Gamma'_i \subseteq \mathsf{br}(U)$ implies that $\mathsf{Lift}(\Gamma'_i, i+1) \subseteq \mathsf{br}(U)$ (since, by Lemma 36, $\mathsf{br}(U)$ is upward-closed), and hence $\Gamma'_{i+1} = \mathsf{br}(\mathsf{Lift}(\Gamma'_i, i+1), i+1) \subseteq \mathsf{br}(U)$. Thus the second containment of (4.9) holds for $i+1$. Also, since $U_{\leq i} \subseteq \Gamma_i$ and $i \geq \mathsf{bw}(U)$, we have

$$\begin{aligned}
\Gamma_{i+1} &= \mathsf{br}(U_{\leq i+1}, i+1) \\
&= \mathsf{br}(\mathsf{Lift}(U_{\leq i}, i+1), i+1) \\
&\subseteq \mathsf{br}(\mathsf{Lift}(\Gamma_i, i+1), i+1) \\
&\subseteq \mathsf{br}(\mathsf{Lift}(\Gamma'_i, i+1), i+1) \\
&= \Gamma'_{i+1}
\end{aligned}$$

Therefore the first containment of (4.9) holds for $i+1$.

Now let N be the final value of the variable n in the unoptimized algorithm. Then, by Theorem 15, we have $\uparrow\!\Gamma_N = \mathsf{br}(U)$, which, along with (4.9) implies $\uparrow\!\Gamma'_N = \mathsf{br}(U)$. Since $\Gamma'_k \subseteq \Gamma'_j$ when $k \leq j$, it follows that for each $i \in \{N+1, \ldots, N+\delta\}$ we have

$$\Gamma'_i = \mathsf{Lift}(\Gamma'_{i-1}, i) = \mathsf{br}(U)_{\leq i} \tag{4.10}$$

Therefore the optimized version also terminates with the final value of n being N, and computes Γ'_N such that $\uparrow\!\Gamma'_N = \mathsf{br}(U)$. $\square$

4.3.5 Near Necessity of the Convergence Theorem

We conclude this section with a note on our Convergence Theorem (Theorem 15). The theorem can be intuitively described as stating that if we are stable for δ consecutive values of i, then we never will find any states outside of the upward closure of the current set of backwards reachable states, regardless of how high we take i. One may wonder if this is overkill; perhaps being stable just a single time is sufficient for convergence. In other words, one wonders if the theorem can be strengthened to say that

$$\mathsf{br}(U, n+1) \subseteq \uparrow\!\mathsf{br}(U, n)$$

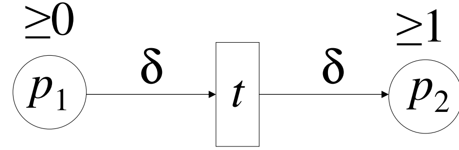


Figure 4.6: The petri net and upward-closed set of markings U discussed in the proof of Theorem 20. U is indicated by constraints above the two places, i.e. p_1 must have at least 0 tokens while p_2 must have at least 1 token. The single transition t may only fire if there are at least δ tokens at p_1, and firing t results in δ tokens moving from p_1 to p_2.

is equivalent to (4.4).

Here we show that such a strengthening cannot be done; there are pathological WSTSs that nearly demonstrate the necessity of the Convergence Theorem. Our demonstration is only *near* for the following reason. To show full necessity, for arbitrary $\delta \geq 1$ one must exhibit a WSTS, an upward-closed set U, and some $n \geq \mathsf{bw}(U)$ such that

$$\mathsf{br}(U, n + \delta - 1) \subseteq \uparrow\mathsf{br}(U, n) \tag{4.11}$$

yet (4.4) fails, i.e.

$$\mathsf{br}(U) \neq \uparrow\mathsf{br}(U, n)$$

Such an example would demonstrate that our algorithm can potentially be stable over all of $\delta - 1$ consecutive values of i, yet then again be unstable on the δth value. We would then conclude that our Convergence Theorem is optimal. Our Theorem 20 below *nearly* achieves such a demonstration; it falls short because we show that a WSTS exists that is stable over $\delta - 2$ iterations, and then is unstable on the $(\delta - 1)$th. This at least shows an asymptotic necessity, and we leave open the problem of whether or not the Convergence Theorem can be slightly improved by replacing (4.3) with (4.11).

Theorem 20. *For each $\delta \geq 2$, there exists a δ-deflatable petri net, an upward-closed set of markings U, and $n \geq \mathsf{bw}(U)$ such that*

$$\mathsf{br}(U, n + \delta - 2) \subseteq \uparrow\mathsf{br}(U, n) \tag{4.12}$$

and

$$\mathsf{br}(U) \neq \uparrow\mathsf{br}(U, n) \tag{4.13}$$

Proof. Taking δ to be fixed but arbitrary, define the petri net (P, T, F) by $P = \{p_1, p_2\}$, $T = \{t\}$, and

$$F((p_1, t)) = \delta \quad F((p_2, t)) = 0$$
$$F((t, p_1)) = 0 \quad F((t, p_2)) = \delta$$

Let $U = \uparrow\{(0, 1)\}$ be the set of markings that puts at least 1 token on p_2. Fig. 4.6 depicts this petri net and U. Note that $\mathsf{bw}(U) = 1$; let us define $n = 1$. It is easy to see that for each $j \in \{1, 2, \ldots, \delta - 1\}$ we have $\mathsf{br}(U, j) = U_{\leq j}$, since the sole transition t cannot fire unless there are at least δ tokens. It follows that $\uparrow\mathsf{br}(U, j) = \uparrow U = U$ for all $j \in \{1, 2, \ldots, \delta - 1\}$, therefore (4.12) holds for $n = 1$. However, $\mathsf{br}(U, \delta) = U_{\leq\delta} \cup \{(\delta, 0)\}$, and noting that $(\delta, 0) \notin U$, we have $\mathsf{br}(U, \delta) \not\subseteq U = \uparrow\mathsf{br}(U, 1)$. This implies (4.13) for $n = 1$. □

4.4 Conjunctive Guard Reduction

Though WSTSs (and indeed NSWs) encompass a broad and important class of infinite state systems, there are common system attributes that preclude well-structuredness. An example of such an attribute is the so-called *conjunctive guard* (CG). CG are used in parameterized systems of processes when a transition is enabled only if the local states of *all* processes satisfy some predicate. This contrasts with petri net or broadcast protocols, in which only a fixed, finite number of processes may guard a transition. Unfortunately, endowing petri nets or broadcast protocols with CG renders even safety property verification undecidable [51]. In this section we develop a sound reduction that reduces a broadcast protocol with CGs to a broadcast protocol without CGs.

BPs were defined in Def. 45; here we extend that definition to define *conjunctively guarded broadcast protocols* (CGBP). A CGBP is a tuple (L, Σ, R, γ), where (L, Σ, R) is a BP, and $\gamma : \Sigma_l \to 2^L$. For each action $a \in \Sigma_l$, $\gamma(a)$ is the *conjunctive guard* of a. The semantics are changed so that a local transition on a may occur only if all other processes are in states that satisfy the conjunctive guard of a.[12] Formally, the semantics of a CGBP are the same as that of a BP (Def. 46),

[12]Our definition of CGBP allows only local actions to have conjunctive guards. The definition and the reduction can be generalized to support conjunctively guarded rendezvous and broadcasts.

except that we strengthen the local transition requirements of Def. 46 as follows.

- *CGBP local transition*: there exists $i \in \mathbb{N}_n$ and an action $a \in \Sigma_l$ such that $(\ell_i, a, \ell'_i) \in R$, and, for all $j \in \mathbb{N}_n \setminus \{i\}$, we have both $\ell'_j = \ell_j$ and $\ell_j \in \gamma(a)$.

A local action a is said to be *properly conjunctively guarded* (PCG) if $\gamma(a) \subsetneq L$. Hence a BP can be viewed as a CGBP in which no actions are PCG, since in this case the additional requirement on each local transition is a tautology.

Our reduction transforms a CGBP $\mathcal{B} = (L, \Sigma, R, \gamma)$ into a BP $\mathcal{B}' = (L', \Sigma', R')$. Intuitively $\mathcal{B}'$ replaces PCG local actions with broadcasts. These new broadcasts allow all processes to check if they *would have* permitted the transition in $\mathcal{B}$, i.e. if their local state satisfies the CG. Whenever a process detects a violation of a CG, it refuses to participate in any future actions by “resigning”; resigned processes are stuck in that state forever.

Formally, we define $\mathcal{B}'$ as follows. We denote by Σ_{pcg} the set of PCG actions in $\mathcal{B}$, i.e. $\Sigma_{pcg} = \{a \mid a \in \Sigma_l \wedge \gamma(a) \subsetneq L\}$.

- $L' = L \cup \{\mathit{resigned}\}$, where $\mathit{resigned}$ is a new local state not in L. A process will enter $\mathit{resigned}$ if it notices (through a broadcast) that a conjunctive guard was violated.

- $\Sigma' = \Sigma'_l \cup \Sigma'_r \times \{!, ?\} \cup \Sigma'_b \times \{!!, ??\}$, where

  - $\Sigma'_l = \Sigma_l \setminus \Sigma_{pcg}$
  - $\Sigma'_r = \Sigma_r$
  - $\Sigma'_b = \Sigma_b \cup \Sigma_{pcg}$

  Thus Σ' is the same as Σ, except that all PCG local actions are turned into broadcasts.

- R' contains exactly the following transitions

  - for each $(\ell, \alpha, \ell') \in R$ such that $\alpha \in \Sigma' \setminus (\Sigma_{pcg} \times \{??, !!\})$ we have $(\ell, \alpha, \ell') \in R'$. Hence all rendezvous, broadcast, and non-PCG local transitions are unchanged.
  - for each $(\ell, a, \ell') \in R$ such that $a \in \Sigma_{pcg}$ we have $(\ell, a!!, \ell') \in R'$. Hence PCG local actions become broadcasts.

– for each $\ell \in L$ and $a \in \Sigma_{pcg}$ we have $(\ell, a??, \ell') \in R'$, where

$$\ell' = \begin{cases} \ell & \text{if } \ell \in \gamma(a) \\ resigned & \text{otherwise} \end{cases}$$

Hence, upon receiving a broadcast corresponding to a PCG action, a process is unaffected if its local state satisfies the conjunctive guard, otherwise it enters $resigned$.

– for each $a \in \Sigma'_b$ we have $(resigned, a??, resigned) \in R'$. These transitions serve only to satisfy the restriction that broadcasts must always be received.

Consider an upward-closed set of states U of a CGBP $\mathcal{B} = (L, \Sigma, R, \gamma)$ and let $B = \mathsf{basis}(U)$ be the canonical basis of U in the subword ordering over L^*. Let U' be the upward closure of B in the subword ordering over the set $(L \cup \{resigned\})^*$. Alternatively, U' can be defined by

$$w \in U' \;\Leftrightarrow\; d(w) \in U$$

where $d(w)$ is w with all occurrences of the symbol $resigned$ deleted. This set U' will be the bad states we wish to avoid in $\mathcal{B}'$.

If $\mathcal{S} = (S, \rightarrow)$ is a transition system, we call a sequence of states $x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_\ell$ a *path* of $\mathcal{S}$. Also, given state sets $I, J \subseteq S$, we say that J *is (un)reachable from* I *in* $\mathcal{S}$ if there exists (does not exist) a path of $\mathcal{S}$ that starts in I and ends in J. The following theorem states that $\mathcal{B}'$ is a sound reduction of $\mathcal{B}$, in that the unreachability of U' in $\mathcal{B}'$ implies that of U in $\mathcal{B}$.

Theorem 21. *Let $\mathcal{B}$ be a CGBP, and let I and U be sets of states of $\mathcal{B}$ such that U is upward-closed. If U' is unreachable from I in $\mathcal{B}'$, then U is unreachable from I in $\mathcal{B}$.*

Proof. By Lemma 40, the set of states reachable from I in $\mathcal{B}'$ over-approximates the set of states reachable in $\mathcal{B}$. Thus, if U' is unreachable from I in $\mathcal{B}'$, then U is unreachable from I in $\mathcal{B}$, since $U \subset U'$. □

Lemma 40. *σ is a path of $\mathcal{B}$ if and only if σ is a path of $\mathcal{B}'$ in which no process is ever in the local state $resigned$.*

Proof. Follows from the simple observations that any transition of $\mathcal{B}$ is a transition of $\mathcal{B}'$, and, conversely, any transition $\mathcal{B}'$ in which no process starts or finishes in $resigned$ is a transition of $\mathcal{B}$. □

Theorem 21 suggests the following verification approach. In order to verify that a CGBP $\mathcal{B}$ cannot reach an upward closed set U from the initial states I, one constructs $\mathcal{B}'$ and checks that U' is unreachable from I in $\mathcal{B}'$. Of course, decidability of this proposition about $\mathcal{B}'$ depends on the form of I; if I is a parameterized set then we have decidability. Similar to our verification approach of Chapter 3, this approach to verifying CGBPs is *incomplete*, meaning that if we find that U' *is reachable* from I in $\mathcal{B}'$, we cannot make any conclusions about the correctness of $\mathcal{B}$.

Our CG reduction is complete, however, if a certain decidable side condition holds. For each PCG local action a of the CGBP $\mathcal{B} = (L, \Sigma, R, \gamma)$, let $\widehat{\gamma}(a) \subseteq L$ be the set of local states ℓ such that there exists a sequence of zero or more non-PCG local transitions taking ℓ to a state $\ell' \in \gamma(a)$; note that $\gamma(a) \subseteq \widehat{\gamma}(a)$. We construct a broadcast protocol $\mathcal{B}''$ that modifies $\mathcal{B}'$ as follows. A new local state fail is added, and when a process in local state ℓ receives the broadcast $a!!$ where $a \in \Sigma_{pcg}$ (i.e. a broadcast corresponding to a PCG action in $\mathcal{B}$), its next local state ℓ' is given by

$$\ell' = \begin{cases} \mathit{fail} & \text{if } \ell \notin \widehat{\gamma}(a) \\ \mathit{resigned} & \text{if } \ell \in \widehat{\gamma}(a) \wedge \ell \notin \gamma(a) \\ \ell & \text{if } \ell \in \gamma(a) \end{cases}$$

As before, the process remains in the same state if it satisfies the CG. Otherwise, if it could have satisfied the CG by making (non-PCG) local transitions only, then it resigns. However, if the process could not have reached a state in the CG "on its own", then it enters fail.

Let Fail be the set of all finite words w over $L \cup \{\mathit{resigned}, \mathit{fail}\}$ such that w contains at least one occurrence of fail. Note that Fail is clearly upward-closed. The following theorem states that if Fail is unreachable in $\mathcal{B}''$, then our conjunctive guard reduction is both sound and complete.

Theorem 22. *Let $\mathcal{B}$ be a CGBP, and let I and U be sets of states of $\mathcal{B}$ such that U is upward-closed, and suppose Fail is unreachable from I in $\mathcal{B}''$. Then U' is reachable from I in $\mathcal{B}'$ if and only if U is reachable from I in $\mathcal{B}$.*

Proof. ($\Leftarrow$) Follows by Theorem 21.

($\Rightarrow$) Since in all three transition systems we are interested in the same set of initial states I, the term *(un)reachable* will always implicitly be with respect to I in this proof. We assume that U' is reachable in $\mathcal{B}'$ and that Fail is unreachable in $\mathcal{B}''$, and show that U is reachable in $\mathcal{B}$. Call a path *unsafe* if it starts in I and ends in U or U'.

For a path σ of $\mathcal{B}'$ and a process p, let $\mathsf{rsl}(\sigma, p)$ be the length of the suffix of σ in which p is in state $\mathit{resigned}$. Note that $\mathsf{rsl}(\sigma, p) = 0$ precisely in the case that p never resigns. Letting n be the number of processes involved in σ, we show how, given an unsafe path σ of $\mathcal{B}'$, we can construct another unsafe path τ of $\mathcal{B}'$ such that

Property 1 for all $p \in \mathbb{N}_n$ we have $\mathsf{rsl}(\tau, p) \leq \mathsf{rsl}(\sigma, p)$, and

Property 2 there exists $q \in \mathbb{N}_n$ such that $\mathsf{rsl}(\tau, q) < \mathsf{rsl}(\sigma, q)$,

By iterating this construction, one can transform any unsafe path of $\mathcal{B}'$ into an unsafe path of $\mathcal{B}'$ in which no process resigns, which, by Lemma 40, is an unsafe path of $\mathcal{B}$.

We now describe the construction. Suppose we have an unsafe path of $\mathcal{B}'$

$$x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_k \tag{4.14}$$

in which some process resigns. Let q be an earliest resigning process, i.e. q is such that there exists $i \geq 1$ where $x_{i-1}(q) \neq \mathit{resigned} = x_i(q)$, and further no process is resigned in x_{i-1}. Then there exists $a \in \Sigma_{pcg}$ (i.e. a is a PCG local action of $\mathcal{B}$ and hence a is a broadcast action of $\mathcal{B}'$) such that $x_{i-1} \rightarrow x_i$ is a broadcast transition on the action a. Since Fail is unreachable in $\mathcal{B}''$, we have that $x_{i-1}(q) \in \widehat{\gamma}(a)$. Thus there exists a path of $\mathcal{B}'$

$$x_{i-1}^0 \rightarrow x_{i-1}^1 \rightarrow \cdots \rightarrow x_{i-1}^m \tag{4.15}$$

such that

- $x_{i-1}^0 = x_{i-1}$,
- for each $j \in \{1, \ldots, m\}$, $x_{i-1}^{j-1} \rightarrow x_{i-1}^j$ involves a local transition of process q, and
- $x_{i-1}^m(q) \in \gamma(a)$

Hence x_{i-1} and x_{i-1}^m differ only in the local state of q, and therefore the broadcast of a may occur from x_{i-1}^m, yielding a state y_i which differs from x_i only in that $y_i(q) = x_{i-1}^m(q) \neq \mathit{resigned} = x_i(q)$. Note that no process is resigned in any state of (4.15). Starting from y_i, there exists a path of $\mathcal{B}'$

$$y_i \rightarrow y_{i+1} \rightarrow \cdots \rightarrow y_k \tag{4.16}$$

such that for each $j \in \{i, \ldots, k\}$, y_j differs from x_j in (at most) the local state of process q. For each $j \in \{i+1, \ldots, k\}$, the transition $y_{j-1} \to y_j$ involves the same local, rendezvous, or broadcast action as $x_{j-1} \to x_j$, which is always possible because $x_{j-1}(q) = \mathit{resigned}$ and thus q never performs an output in these transitions. Intuitively, in (4.16) q is passive in the sense that it only receives inputs from broadcasts, possibly resigns at some point, but is definitely not resigned in y_i. Since replacing any occurrence of $\mathit{resigned}$ in a word of U' with any other element of L will yield another word of U', and $x_k \in U'$, we have $y_k \in U'$.

We may construct the desired path τ by concatenating the first i states of (4.14), all of (4.15), and all of (4.16), as follows.

$$x_0 \to x_1 \to \cdots \to x_{i-1} \to x_{i-1}^1 \to \cdots \to x_{i-1}^m \to y_i \to \cdots \to y_\ell \tag{4.17}$$

It is easy to see that (4.17) is unsafe and satisfies properties 1 and 2 above, hence this completes the proof. □

4.5 Evaluation

In this section we describe our implementation of our algorithm (Sect. 4.5.1) and then provide an informal comparison with the classical algorithm (Sect. 4.5.2). The remaining subsections present experimental results for several petri nets (Sect. 4.5.3), for a MESI cache protocol over multiple cache blocks (Sect. 4.5.4), and for a more elaborate caching protocol (Sect. 4.5.5). The latter two systems both involve properly conjunctively guarded transitions; our CG reduction of Sect. 4.4 was successfully applied in both cases. All experiments were run on a machine with an Intel Pentium 4 at 2.6GHz and 4GB total memory. The implementation of the classical approach we compare against is based on an extension of CSTs called *interval sharing trees* [59].

4.5.1 Implementation

Our proof-of-concept implementation assumes a NSW in which states are words over a finite alphabet L, and the wqo is the subword ordering $\preccurlyeq_{\mathsf{W}}$. As mentioned in the proof of Theorem 14, in this ordering, the weight of a word is simply its length. Given a binary encoding of L, we can naturally represent a subset of L^n for some $n \geq 1$ using a BDD [23]. The ensuing discussion is simplified if

we pretend that the BDD variables are $x_1, x_2, \ldots$, where each x_j ranges over L; of course in reality each x_j corresponds to many boolean variables that encode elements of L.

For each $i \geq 1$, our implementation requires an SMV description of both the transition relation for the system restricted to i processes and the bad states involving i processes. These descriptions are generated beforehand up to a small guessed maximum value of i. Also, the Lift operation is expressed by an SMV transition relation such that the postimage of the relation is the result of applying Lift; (4.19) below demonstrates how this is done. Given these SMV descriptions, line 7 of our algorithm (Fig. 4.4) or the optimized version $7'$ of Sect. 4.3.4 is computed using standard finite state backward reachability analysis. This yields a BDD representation of Γ_i.

Line 8 of Fig. 4.4 checks if the initial states have been reached. Here we explain how our implementation handles the generalization of Sect. 4.3.3 for infinite parametric sets. Thus our initial states are expressed by (4.6) and we must decide (4.7) for each i. This is achieved by building the BDD for:

$$\Gamma_i \wedge \left(\bigwedge_{j \in E} \quad \bigvee_{K \subseteq \mathbb{N}_i \,\wedge\, |K| = \max(i - a_j, 0)} \quad \bigwedge_{k \in K} x_k \neq \ell_j \right) \tag{4.18}$$

It is straightforward to show that (4.7) holds if and only if (4.18) is not the empty BDD, i.e. false. Though the computation of the BDD (4.18) has the potential to blow up, in practice this is not a huge concern, since typically $|E|$ is small, as is a_j for all $j \in E$.

Line 11 involves an application of the Lift operator and a set containment check, the latter is decided simply using BDD implication. The Lift operation is computed as[13]

$$\mathsf{Lift}(\Gamma_{i-1}, i) \quad = \quad \bigvee_{k=1}^{i} \mathsf{rename}_k(\Gamma_{i-1}) \tag{4.19}$$

where, for each $k \in \mathbb{N}_i$, $\mathsf{rename}_k(\Gamma_{i-1})$ is simply Γ_{i-1} with the following variable renamings: for each $j \in \{k, \ldots, i-1\}$ we rename x_j with x_{j+1}. Of course $\mathsf{rename}_k(\Gamma_{i-1})$ will be isomorphic to (and hence exactly the same size as) the BDD for Γ_{i-1}; the potential for BDD blow up stems from the i-way disjunction of (4.19).

[13] In our make-believe world where BDD variables range over L, (4.19) is accurate. However in reality, where variables are boolean, we must prevent the disjunction from introducing variable assignments not corresponding to elements of L. This is handled by simply conjoining a BDD for L^i to the RHS of (4.19).

4.5.2 Comparison to Standard Approach

Here we provide an informal comparison between our algorithm and the classical algorithm, specifically the implementation of the classical algorithm based on CSTs [42, 43] which was briefly discussed on page 102.

CSTs are similar to BDDs in spirit, however, the fact that semantically a CST denotes the *upward-closure* of its constituent vectors complicates the manipulation algorithms. For example, checking subsumption (i.e. upward-closed set containment) between two CSTs is co-NP hard in their sizes. Unfortunately, checking subsumption is an integral part of the classical algorithm (cf. the condition of the while-loop in Fig. 4.2). To combat this problem, Delzanno et al. develop a sophisticated heuristic solution in which certain CST simulation relations facilitate pruning of an (exponential time) exact subsumption check [43]. In contrast, subsumption between two BDDs can be decided in time proportional to the product of their sizes [23]. In fact, we can correctly replace the containment of line 11 of Fig. 4.4 with an equality test: $\Gamma_i \neq \mathsf{Lift}(\Gamma_{i-1}, i)$. This test can be done *in constant time* using a reasonable BDD library, such as CUDD [119]. On the other hand, the potential for BDD blow-up, especially in computing the Lift operator à la (4.19), is always a looming concern.

The main efficiency difference between the two approaches is likely to derive from the sizes of the underlying data structures. Predicting the dynamics of these sizes is a complex problem. Though BDDs compactly represent many practical boolean functions, the worst case size is exponential in their height (i.e. the number of boolean variables). Similarly, although bounds on the size of CSTs have not been derived in the literature (to our knowledge), any such bound is at least exponential in the height of the structure. Here, we consider data structure height as a coarse measure of worst-case size.

The CST-based classical algorithm assumes that states of the WSTS are vectors of $\mathbb{N}^m$, and can be applied to both petri nets and broadcast protocols. Let L be the set of local states (in the case of broadcast protocols) or the set of places (in the case of petri nets). Then we call $|L|$ the *dimensionality* of the system. The height of the CSTs is fixed and equal to the dimensionality, while if we employ L^* as the state space and use our algorithm, the height of the involved BDDs is at most $(n_f + \delta) \lceil \log_2 |L| \rceil$, where n_f is the final value of n in our algorithm. This suggests that our approach might be superior when $(n_f + \delta) \lceil \log_2 |L| \rceil$ is much less than the dimensionality, since

Petri net	Our runtime	CST runtime	Max BDD height	CST height (dimensionality)
Multipool	3010	2.09	50	18
CSM	95	0.06	36	14
Mesh(2 × 2)	>1300	1.30	>40	32

Table 4.2: Experimental results for selected petri nets from a paper of Delzanno et al. [43]. For Mesh(2 × 2), our implementation spaced out.

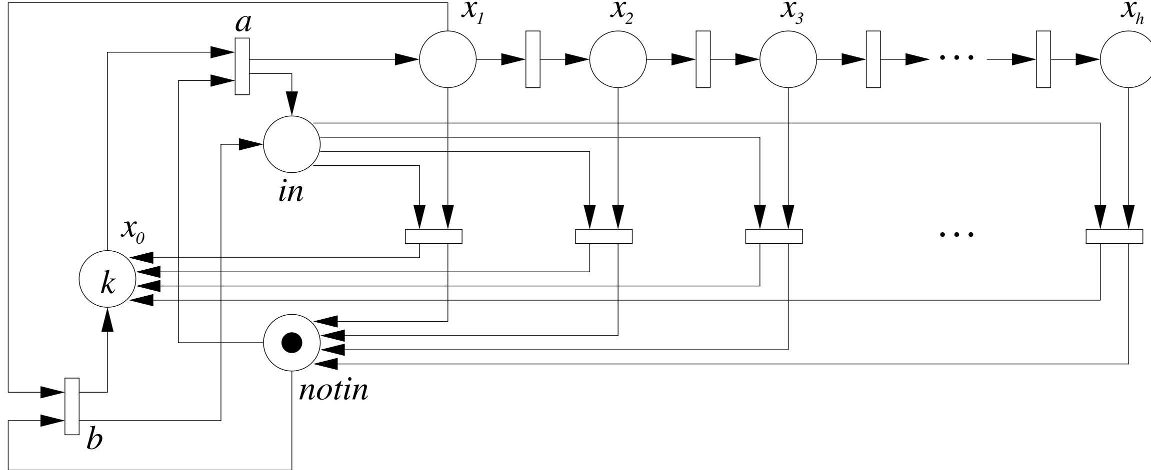


Figure 4.7: The petri net $ME(h)$

under such circumstances the CSTs are more likely to blow-up. Of course predicting n_f a priori is likely intractable. However, $|L|$ and δ *are* known a priori, and hence even an estimated value of n_f could be used to determine which approach to apply.

We now present our experimental results. For those where both approaches are used, the results tend to confirm our hypothesis that data structure height is a good predictor of performance.

4.5.3 Petri Nets

Delzanno et al. [43] have run their CST-based implementation of the classical approach against several petri nets. These nets have small dimensionality, so, as discussed in Sect. 4.5.2, we do not expect our approach to perform well. Indeed, Table 4.2 shows that the CST-based implementation outperforms our approach by several orders of magnitude. Recall that we anticipate our approach to have an advantage when the height of our BDDs is dwarfed by the height of the CSTs, which is not the case here. In fact, for all three petri nets, the CSTs enjoy a shorter height than the BDDs.

In hopes of confirming our hypothesis about data structure height, we constructed the "param-

eterized petri net" $ME(h)$ depicted in Fig. 4.7. The parameter h dictates the width of the chain of places $x_1, \ldots, x_h$ along the top of the figure. h allows us to control dimensionality, which is equal to $h+3$. For any h, the parametric set of initial markings for $ME(h)$ places a single token at the place $notin$, and an arbitrary number k on x_0. A token can move from x_0 to x_1 when there is a token at $notin$ by firing transition a. This will move the token at $notin$ to in, which will disallow any more tokens from entering x_1. The token that is in the chain can move along the chain, and at any point it may hop back to x_0, which will result in the token at in being moved back to $notin$. Hence $ME(h)$ implements mutual exclusion in the sense that at most one token can be in the chain at any time. For our experiments, we verified that there can only be at most one token at x_h; i.e. the upward-closed set of markings $\{m \mid m(x_h) \geq 2\}$ is not reachable.

We measured execution times for $h = 25, 50, \ldots, 250$; the results are plotted in Fig. 4.8. The plot labelled "CST" gives the runtime for the Delzanno et al.'s CST approach [42, 43].[14] The plot labelled "BDD" gives the total runtime for our implementation *and* a translation step to take the original petri net description to the SMV required of our tool; "BDD minus translation" disregards this translation time. The latter is presented because the translation phase is suboptimal; in a sophisticated implementation there would be no need for the intermediate SMV files and hence the time in this phase would be highly mitigated.

From Fig. 4.8 we see that for $ME(h)$ our algorithm is almost two orders of magnitude faster than the state-of-the art implementation of the standard approach (for large h). The probable trend for $h > 250$ is clear. With regard to data structure height, we note that for $ME(250)$, the CSTs are a towering 253 levels high, while the largest BDD processed by our algorithm had a height of merely 48.[15]

4.5.4 MESI Protocol

MESI is a common variety of shared-memory protocol. In a MESI protocol, each client has a cache block in one of four states: *Modified*, *Exclusive*, *Shared*, or *Invalid*. Although many MESI protocols are conceivable, here we use the standard version used by computer architects (as presented by

[14] In fact, the reported run-times are those of software based on *interval sharing trees* (IST) which are an extension of CSTs to handle two-sided constraints [59]. The IST software used is a constant 3 or 4 times slower than a pure CST implementation.

[15] Here we found $n_f = 4$, $\delta = 2$, and $|L| = 253$.

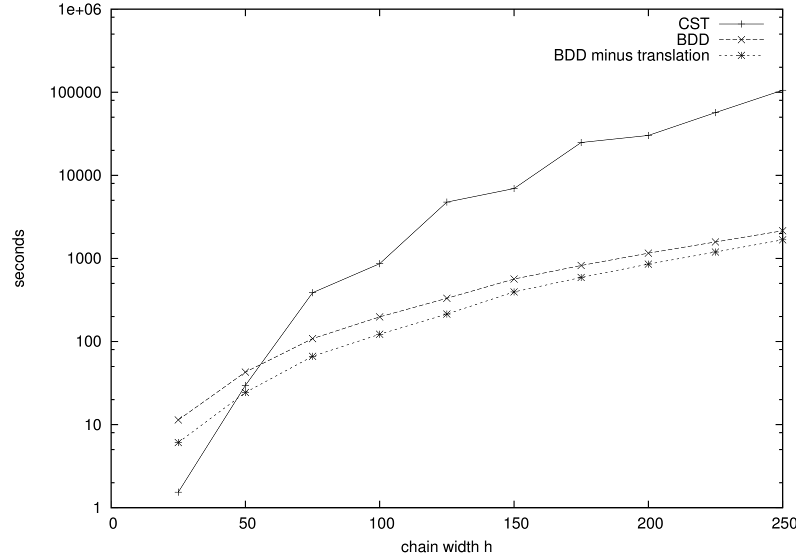


Figure 4.8: Execution times for the two approaches against the petri net $ME(h)$ of Fig. 4.7. The vertical axis gives the runtime in seconds; the horizontal axis gives the petri net parameter h. Our approach is labelled “BDD”, while the CST approach of Delzanno et al. is labelled “CST”.

Culler et al. [38]), which has a PCG transition, so we use our reduction from Sect. 4.4.

As for $ME(h)$ of Sect. 4.5.3, we desired a “knob” that would give us control over the dimensionality. Since shared-memory protocols are used to orchestrate the sharing of multiple addresses, we instantiated the MESI protocol over m addresses,[16] for $m \in \{1, 2, 3, 4\}$.

The results are given in Table 4.3. Our CG reduction allowed the verification to succeed, so there was no need to verify the side condition. We compare our result against both the CST-based classical approach and another variant of the classical approach based on the polyhedral model checker Hytech [40, 73]. The results clearly indicate the superior scalability of our approach as the local state (dimensionality) grows. The Hytech-based approach aborts even for $m = 3$, reporting “Out of memory”. The parser of the CST tool cannot handle the size of the description of MESI with 4 blocks, which is 5.9 MBs. This large size arises because the broadcast matrices used by the classical approach grow quadratically in the dimension of the problem. This contrasts with the mere 30 KB of SMV that constitutes our tool’s input.

[16]In this case, since the “sub-protocols” controlling each address are independent, correctness for $m = 1$ entails correctness for all $m \geq 1$. In practice, however, such a simplification is often not possible, because real protocols can exhibit nontrivial interactions between different addresses. This experiment measures how our approach handles the explosive growth in local state resulting from analyzing multiple addresses.

# of blocks	Our runtime	CST runtime	Hytech runtime	Max BDD height	dimension
1	0.0	0.0	0.0	9	5
2	0.1	0.2	380.0	18	25
3	0.7	131.9	$>$7989.0	27	125
4	4.6			36	625

Table 4.3: Results for the MESI protocol with PCG over multiple blocks. Run times are in seconds. The column *dimension* indicates the height of the CST data structures in the CST approach, and also the width of the real vectors processed by Hytech in Delzanno's polyhedral approach.

Property (all passed)	Runtime (sec)
Encoding of curPtr	3
CG reduction completeness	214
Data coherence	63

Table 4.4: Results for German's Protocol. To convert the protocol to a CGBP, we needed to re-encode the curPtr variable. Although this encoding was straightforward, we verified that encoding is one-hot as a sanity check. The main verification task was "data coherence", which verified that the value of each read is the most recently written data value. Since the CG reduction is sound and the verification succeeded, we actually did not need to run the "CG reduction completeness" verification task, which performs the completeness check suggested by Theorem 22. We have provided the run time simply to illustrate that the side condition is verifiable in practice.

4.5.5 German's Protocol

German's protocol [62] is a challenge problem for parameterized verification that has been previously tackled in several papers [28, 86, 109]. As done by Chou et al. [28], we include a one bit data path. The original description [62] is a Murφ model, and is almost a CGBP: the Murφ description involves a variable that is a pointer to a process; such variables are not directly supported by CGBPs. We've encoded this variable by simply giving each process an extra bit, which is true at a process iff the original variable would point to the process. This system is a CGBP, to which we applied our CG reduction to obtain a broadcast protocol.

As mentioned in Sect. 4.5.4, even *describing* a BP in a format suitable for the classical approach is problematic when the dimension is large. Due to its various channels and the presence of data variables, our model of German's protocol has a dimensionality of 6144. For this reason, we were unable even to run the CST-tool against this example. The results for our tool are given in Table 4.4.

Chapter 5

Related Work

In this chapter we survey previous research on verification of shared memory protocols, memory models, and parameterized verification.

In Sect. 5.1 we look at early efforts to model check shared memory protocols that tackle safety properties (e.g. at all times, at most one processor has exclusive permissions to a block) or, more generally, properties expressed in some standard temporal logic. Typically, these properties are chosen to imply, or at least increase confidence, that the system adheres to the intended shared memory model, but the memory model is not verified per se.

Like SC, most memory models are defined in terms of the existence of certain reorderings of the memory traces, notions that typical temporal logics are too weak to capture.[1] In Sects. 5.2 and 5.3, we summarize works that *do* strive to verify the memory model directly. Sect. 5.2 summarizes approaches involving hand-proofs and theorem provers, while Sect. 5.3 considers memory model verification that involves model checking.

Techniques involving hand proofs or theorem proving typically verify correctness across all configurations of interest, of course this thoroughness comes at the expense of human labor. Indeed, many of the papers of Sect. 5.2 prove correctness over all parameterizations. Sect. 5.4 looks at parameterized verification research that is automated or at least semi-automated. Under this partitioning of the literature, contributions that do (automated) parameterized verification with respect to a memory model could either be listed in Sect. 5.3 or 5.4; we have elected to include them in the latter.

[1] An obvious counterexample here is Lamport's TLA [89], which is undecidable.

The final Sect. 5.5 mentions several avenues of related that do not fall under any of the previous four section headings.

5.1 General Protocol Verification

Shared memory protocol verification was one of the early successes of symbolic model checking when SMV was employed by McMillan and Schwalbe [97] to the industrial Gigamax multiprocessor memory system, by Clarke et al. [30] to the Future+ cache coherency protocol (an IEEE standard), and by Eiríksson and McMillan [48] to the SGI Origin 2000 protocol. All three verification efforts revealed several previous unknown defects in the respective designs. Explicit state model checking was also shown to be effective at protocol analysis by Dill et al. [46], who developed the Murφ language and model checker. Later, Hu et al. [78] applied Murφ to another industrial shared-memory system, carefully tracking human effort, and concluded that such endeavors are worthwhile from an economic standpoint.

5.2 Memory Model Verification: No Model Checking

Here we discuss works wherein the specification is a memory model, but do not use model checking as the primary means of verification. Some approaches are hand-proof methodologies or case-studies, others are unsound but useful for catching bugs, and others are formally sound thanks to a automated theorem prover, but still require a large component of human effort.

Perhaps the earliest effort that turned attention to testing adherence to a memory model was that of Collier [33]. The work, which formalized the memory event abstraction used in this thesis and elsewhere, spawned a set of software tests called ArchTest [32] that test a multiprocessor for conformance to various ordering *rules*. Of course, these tests can only *reveal defects* rather than *prove correctness*. Nevertheless, ArchTest has been beneficial enough to both IBM and Intel to warrant worldwide licensing by these industry leaders.

Proving that the Lazy Caching protocol is SC was the subject of a special issue of the journal *Distributed Computing* [99], which contains six different hand proofs. As highlighted in Merritt's introduction, the proofs encompass a broad range of techniques: automata, temporal logics, process algebras, transducers, and refinements. None of the proofs use any automation or are mechanically

checked. However, Graf's proof [71] involves a set of $\forall$CTL* formulas that together imply SC; these formulas are broad enough to verify Lazy Caching. Thus in principle one could model check these properties and infer SC of a given finite state instantiation of the protocol.

Condon et al. [34, 108] have employed a hand-proof methodology based on Lamport clocks [87] to prove SC, as well as the weaker models TSO and Alpha. This framework involves the human (on paper) time-stamping memory events with logical time triples; these timestamps must respect temporal causality and are used to prove the existence of the necessary reordering of events. The same group have applied their technique to prove SC of a more complex directory protocol (very similar to the WIS protocol we experimented with in Sect. 3.6), and have developed an intuitive, table-based protocol specification methodology [120].

Several major efforts to verify memory model adherence using Lamport's *temporal logic of actions* (TLA) have been conducted [7, 90]. The goal of these projects was to verify Compaq's complex EV6 (a.k.a *wildfire*) and EV7 multiprocessor memory protocols, which implement the Alpha memory model. Manual TLA proofs that the protocols implement an operational specification of the Alpha memory model, also written in TLA, were attempted. Although only the proof for the later and simpler EV7 was completed, deep bugs were found in both protocols. Further, an inconsistency between the EV6 implementation and the memory model exposed by the project was ultimately deemed to be an error in the memory model itself; this induced a change in the Alpha Architecture Reference Manual. TLA was also used to construct a detailed hand-proof that Lazy Caching is SC [85]. The TLC model checker [127] was reported to be a valuable tool in the EV7 and Lazy Caching verification projects.

The PVS theorem prover was employed by Arons [11] to verify SC of both Lazy Caching [6] and a ring protocol of Collier [33]. The approach is based on ideas from the original hand proof for Lazy Caching [6], but is mechanically checked in PVS. The use of theorem proving allowed for parameterization over processors, addresses, values, and queue depths. Strangely, for Lazy Caching, Arons proves SC with respect to the internal MemoryWrite events, rather than the usual W events that occur on the protocol's interface (see our Fig. 2.3). Viewed in this light, Lazy Caching looses its status of being SC and not simple SC (see our Def. 27), an attribute that makes the protocol a challenge to verify. Other papers that employ PVS are those of Park and Dill [105, 106], who define a compelling technique called *aggregation of distributed transactions* to verify the FLASH

protocol, and Stoy et al. [123], who use a TLA implementation in PVS and also handle liveness. Park and Dill's work verifies that FLASH (in "delayed" mode) is SC by showing that it is serial.

Related to our Sect. 2.4, there are several papers that consider the question of whether a given trace adheres to a some memory model. Analogous to our result that checking a trace for DSC is NP-complete, Gibbons and Korach [66] have proven the same result with respect to SC. Furthermore, they consider many restrictions of this problem, for example traces involving at most 3 processors, traces in which no processor performs more than 2 events, and traces involving only 2 addresses; all of these remain NP-complete. Our reductions of Sect. 2.4 and Sect. 2.6 are in the spirit of this work. Later, Cantin [26] showed that the problem is NP-complete for a *single* address, i.e. the trace problem for *per-address* SC[2] is NP-complete. Gopalakrishnan and Yang et al. [70, 126] have tackled the problem of making practical tools for solving the trace problem with respect to an arbitrary memory model. Their work has explored techniques of compiling a trace/memory model pair into a boolean satisfiability instance, a quantified boolean formula instance, or a constraint programming problem, which are satisfiable if and only if the trace is admittable under the memory model.

In this thesis, SC (and DSC, etc.) are taken to be properties of finite traces. Glusman and Katz [67] have studied the implications of extending SC to a property of infinite traces. They show that SC is not a safety property by giving an example of an infinite trace that *is not* SC, yet all finite prefixes *are* SC.[3] Hence, one cannot infer that a protocol is SC in the infinite sense by verifying that it is SC in the finite sense. To remedy this situation, Glusman and Katz propose a reasonable set of protocol conditions that are sufficient for the above inference to hold. Considering SC as a property of infinite traces essentially adds an aspect of liveness to the property. Sezgin, who's work is discussed in the next section, also amalgamates liveness properties with SC.

5.3 Memory Model Verification: Model Checking

This section considers model checking-based approaches to verification of various memory models.

[2]per-address SC is called both *coherency* and *cache consistency* throughout the literature

[3]The infinite trace is (essentially) $(W, p_1, x, 1)(W, p_1, x, 2)(R, p_2, x, 1)^\omega$, i.e. $(R, p_2, x, 1)$ is repeated ad infinitum. For any finite prefix, a serial reordering is obtained by simply placing $(W, p_1, x, 2)$ after all events of p_2. However, the whole infinite trace does not have a serial reordering, since wherever $(W, p_1, x, 2)$ is placed, there will be an infinite number of instances of $(R, p_2, x, 1)$ after it.

Nalumasu et al.'s *test model checking* [104] is a fusion of the ArchTest rules with model checking. Test model checking is sound and complete for several memory models, however, the approach is unsound for SC [68]. In his thesis, Nalumasu [103] pioneered much of the constraint graph theory we employ in Chapter 3. Ghughal et al. [65] subsequently concocted new ArchTest-style rules for several weak memory models, and showed how the test model checking paradigm could be used to test these properties.

Park and Dill have implemented *maximally general* models for the SPARC architecture memory models (TSO/PSO/RMO) using Murφ [45, 107]. The goal of this research is to construct an executable specification which can be used to verify that a given parallel program fragment is correct under a memory model, as opposed to verification that a given protocol hardware design implements the model. Our protocol $\mathcal{G}_k(n, m, v)$ defined in Sect. 2.5 is also an example of a maximally general model, since it completely captures the traces of DSC$_k$.

A verification technique is proposed by Chatterjee and Gopalakrishnan [27] that exploits a natural architectural partitioning found in most modern memory system designs. The approach replaces the so-called *external partition* (consisting of caches, interconnect, and main memory) with an abstracted, simplified structure. The resulting system must be shown to be both refined by the implementation and correct with respect to the memory model. The latter task is facilitated by splitting single write events into local and global instances. Handling of differing logical and temporal orders is done in an ad hoc manner.

A recent and fresh look at the problem of memory model verification has been taken by Sezgin [117, 118], who has developed an alternate means of both defining SC and modeling protocols. Sezgin is critical of the standard trace-based approach to modeling protocols taken in this thesis and elsewhere, and advocates a transducer-based model. These transducers distinguish protocol inputs (i.e. memory access instructions) from outputs (i.e. responses to the instructions). Both alphabets are identical to our set of memory actions, with the exceptions that symbols are annotated according to whether they are inputs or outputs, and input reads do not have a data value component. This scheme allows for making assertions not only about reorderings of events, but relationships between inputs and outputs. One such requirement, which is reminicient of our disallowing of "prophetic inheritance" in DSC (see Sect. 2.1), is that a data value d may be read from an address a via a read response only after some write instruction has written d to a. Also, protocols are defined to have the

desirable properties that any sequence of input instructions is accepted, and all inputs are eventually responded to with outputs. Whereas in the trace-based formalism a system with the empty set of behaviors is trivially SC, Sezgin precludes such systems from even achieving the status of being legal implementations.

His perspective contrasts with a viewpoint implicitly taken in this thesis: SC is a safety property, and only one of several properties that a correct shared memory protocol is possibly expected to possess. Philosophically, we feel that formal verification should strive to dissect properties as much as possible, rather than amalgamate them into monolithic correctness criteria. However, if a certain amalgamation can be shown to reduce verification complexity or shed light on the difficult problem of shared memory protocol verification, then clearly it is a worthwhile endeavor.

Perhaps the most salient parallel between the work of Sezgin and this thesis are his class of finite state protocols[4] $SC_{\mathcal{P},\mathcal{C}}(j,k)$ and our class $\mathcal{G}_k(n,m,v)$ of Sect. 2.5. Both serve as maximally general protocols from which decidability results are obtained by appealing to automata language containment. We feel that (modulo the differing transducer/trace formalisms used) our class DSC$_k$ captured by $\mathcal{G}_k(n,m,v)$ is more expressive (with respect to reorderings admitted) than the $SC_{\mathcal{P},\mathcal{C}}(j,k)$ machines. Behaviors of the latter have the property that the "distance" in logical time (i.e. the reordering) that any two processors can become out of sync is bounded; this bound is a function of j, k, and $|\mathcal{C}|$. At the level we model protocols, any non-serial SC protocol with which we are familiar does not have such a bound. Indeed, Sezgin's thesis [117] shows that only a *fair* version of Lazy Caching (see Sect. 2.3.3) has behaviors contained in some $SC_{\mathcal{P},\mathcal{C}}(j,k)$. The notion of fairness called upon is much more restrictive than the usual notion. For example, whenever the queue out_i is nonempty, it is required that a MemoryWrite$_i$ event must occur within a bounded number of protocol transitions.[5] In contrast, in Theorem 3 we argue that the unadulterated Lazy Caching is DSC$_k$ for some k (a function of the number of processors and queue depths).

We close our summary of Sezgin's work by mentioning that, due to the fundamental differences in the frameworks, the undecidability results of Alur et al. [8] and this thesis (Sect. 2.7) do not directly apply to his notion of SC. Indeed, in a recent paper he asserts that "the decidability of

[4]Of course his notion of *protocol* differs from ours; his is a finite state *transducer* while our is a finite state *automaton*.

[5]Whereas usual notions of fairness would only require that out_i being nonempty implies that the event MemoryWrite$_i$ occurs *eventually*, with no imposed bound on when the event transpires.

checking the sequential consistency of a finite-state memory system is still an open problem" [118, page 7].

The work of Braun et al. [22] presents a semi-automated SC verification technique that uses model checking. The idea is that the user annotates the protocol description to maintain auxiliary state called a *window*, which summarizes the intended reordering of the trace. When model checking, a finite state automaton called a *checker* is composed with the augmented protocol. The checker verifies that all updates to the window follow a specific set of rules that imply SC. Our view windows (VW) of Sect. 2.5.1 are similar to and were inspired by the windows of Braun et al, and could be used in the same manner that windows are employed. VW are somewhat more expressive in terms of reorderings admitted, however. In particular, if a read event r precedes a write event w temporally, then windows insist that r must also preceed w in the reordering. VW admit DSC traces that *must* break this restriction in their reorderings.

Condon and Hu's approach [36] automates a variant of Braun et al.'s method [22]. The protocol is automatically augmented to dynamically construct an abbreviated version of the constraint graph (see Def. 29 for a related definition) of the trace thus far. The approach requires that the protocol has the attributes of *tracking labels* and a *store order generator*. Tracking labels allow on-the-fly inference of the inheritance relation. The store order generator is an automaton which determines the ordering edges between writes to the same address. In the case that the protocol is simple SC (the property we handle in Chapter 3; see Def. 27), the store order generator is trivial. Otherwise, by bounding the number of store order generator states by the cardinality of the state space of the protocol, the method can theoretically still proceed without user intervention, since all of the finite number of possible store order generators can be enumerated and attempted.

Qadeer's work [113] also automatically verifies simple SC,[6] and inherently supports parameterized verification of a value-parameterized protocol family (see our Def. 5). As in Chapter 3, the family must satisfy the assumptions of data independence, processor symmetry, and address symmetry. An appealing aspect of his framework, which is fully automatic, is that the *violating* behaviors are specified by a set of relatively small finite automata. Collectively, these automata dynamically detect the existence of a canonical k-nice cycle (see our Def. 31) in the trace of the protocol. Model checking is then done on the composition of the (undeterminized) automata with

[6] Qadeer [113] refers to simple SC as SC with the *simple witness*.

the protocol, the result of which will have a state space of size at most the product of that of the protocol and the automata. Qadeer's approach was used in this thesis to verify SSC of the abstraction Q in the experiments of Chapter 3. Prior to this work, Qadeer made a more ambitious effort to handle a broad class of memory models (not just SC) in a similar framework [112].

5.4 Parameterized Verification

The term *parameterized verification* refers to any verification endeavor that proves correctness of a family of systems parameterized by one or more quantities. Parameterized verification is a special case of *infinite state verification*, since an infinitude of systems typically entails an infinitude of states. As mentioned in the introduction to this chapter, hand-proofs and theorem proving proofs usually verify correctness over all parameterizations, however the burden on the human expert is large. *Parameterized model checking* (PMC) attempts to alleviate this burden by automating all or most of the parameterized verification process. This section summarizes work on PMC. We note that most PMC efforts target systems in which the parameter specifies the number of *process* components. These are components that are active and have strong interactions with each other, as opposed to components that are less active and usually easier to deal with, such as addresses or data items. Thus the work discussed in this section is more closely related to our Chapter 4 than Chapter 3, a notable exception being a paper by Henzinger et al. [74] discussed below.

Well-structured transition systems (WSTS), which are a central notion of Chapter 4, were first proposed by Finkel [56]. The upward-closed set backward reachability algorithm for WSTS (i.e. what we call the *classical* approach) was first articulated by Abdulla et al. [2], in which the application to petri nets is observed. Later, Finkel and Schnoebelen generalized the notion and provided an assortment of examples [57]. Recently, Geeraerts et al. [61] have proposed a compelling approach to verification of WSTS based on forward reachability. This is the first sound and complete algorithm that performs forward analysis of WSTS. Similar to our approach, theirs is based on a framework in which a sequence of finite-state subsystems of increasing size are examined until either a counterexample is found, or a certain convergence condition is reached. Convergence occurs when an abstraction, which becomes more and more precise, is tight enough to verify non-reachability.

Broadcast protocols (BP) were introduced by Emerson and Namjoshi [53]. Their procedure for verifying BP safety properties, which is based on the Karp-Miller petri net covering graph [81],

was shown to not necessarily terminate by Esparza et al. [55]. Decidability of safety properties for broadcast protocols was established by Esparza et al. [55], who show how the backward reachability algorithm of Abdulla et al. [2] can be applied to this problem. Delzanno et al. have investigated means of symbolically representing upward-closed sets in the backward reachability algorithm [41–43] with encouraging results. Delzanno advances the standard algorithm for broadcast protocols in two ways [40]. First the permitted transition guards are generalized (though this generalization causes the approach to be only semi-algorithmic) , and second, he shows how efficient real (as opposed to integer) constraint solvers can be used to implement the algorithm. Our Sect. 4.5.4 presented some experimental results that use Delzanno's approach.

So-called *cut-off* results are those that reduce the correctness of a class of parameterized system to the correctness of the system with only n processes. The cut-off n is typically a function of the size of a single process. German and Sistla consider CCS style processes and prove exponentially sized cut-offs for regular properties among other PMC results [63]. For protocols with either conjunctive or disjunctive guards, Emerson and Kahlon give cut-offs for checking LTL\X formula [49]. Our approach of Chapter 4 can be viewed as computing the minimal cut-off for the system protocol being model checked, whereas these previous works infer cut-offs for entire classes of systems. Pnueli et al. have developed smaller cut-off results for a more expressive class of systems [12, 109], but this benefit comes at a cost since completeness is sacrificed.

Emerson and Kahlon [51] have also contributed a thorough study of PMC decidability for various combinations of properties and communication primitives, which includes broadcasts, and have proposed a sound *and complete* verification technique for a class of protocols with CG [52]. However it is unclear if the latter approach will scale beyond systems with small local state. For example, their subsequent treatment of German's protocol requires a nontrivial amount of manual reasoning [50].

Network invariants are a semi-automatic approach to PMC [29, 31, 82, 83, 91, 125]. Given a process template P, the goal is to verify that for all n, the composition of n copies of P, denoted $P^n = P||\cdots||P$ (where there are n Ps), satisfies some property ϕ. The user provides a process I called the *network invariant*, that attempts to abstract the behavior on the interface of P^n for arbitrary n. Proof obligations typically involve verifying that $P||I$ is abstracted by I, that P is abstracted by I, and that $I \models \phi$. All three of these obligations can typically be dispatched to finite

state model checking. From these premises one can conclude the verification goal $\forall n : P^n \models \phi$. A drawback of using network invariants is that they are often a challenge to produce. Furthermore, network invariants are not guaranteed to exist in general [3, 125].

The only other approach known to us that uses model checking to verify sequential consistency of address-parameterized families of protocols is that of Henzinger et al. [74]. Their semi-automatic method, is based on network invariants, can handle families of protocols that are parameterized in the number of processors, in addition to the number of addresses and data values, but only works for protocols whose traces can be reordered by a finite state automaton to form serial traces, a restriction that typically holds in practice only for protocols that are already fully serial. A limitation of their method is that its soundness relies on the assumption that the protocol to be verified is *location monotonic*, meaning that any trace of the system projected onto a subset of addresses is a trace of the smaller system involving just those addresses. Henzinger et al. do not provide a method (automated or otherwise) for testing location monotonicity of a parameterized family of protocols, i.e. it is only confirmed by manual inspection [115]. In his thesis, Nalumasu [103] does provide a framework for constructing protocols that guarantees location monotonicity, but the framework seems to have limited expressiveness.

McMillan has employed the SMV proof assistant [95] to verify the Stanford FLASH multiprocessor's shared memory protocol [96] for an arbitrary number of processors. The proof demonstrates that the protocol is per-address SC. Verification proceeds through user-guided model checking; this guidance comes in the form of case splitting, noninterference lemma writing, and inclusion of auxiliary variables. Chou et al. [28] also use noninterference lemmas in the context of the Murφ verifier. These works were discussed at length in Sect. 3.7 of this thesis.

Abstraction [37] has been used to tackle PMC; such approaches are usually incomplete. Several works use replication abstraction to reduce a parameterized system to finite state [80, 110, 111]. These involve abstracting global states by tracking if there are zero, exactly one, or some multiplicity of processes in each local state. Of these three works, only Pong and Dubois [111] verify a (nontrivial) memory model. An interesting aspect of their work is that verification is restricted to parallel programs that have the property *data race free 1* [5], rather than *all possible* programs. This is achieved by having automata (one per processor) restrict the sequence of issued memory events to be data race free 1. The memory model *release consistency* is verified by augmenting data

copies with certain tags; verification checks that these tags are always consistent with protocol state semantics. Predicate abstraction has also been successfully applied to PMC, for example in Das et al.'s "experiences" [39].

Regular model checking [21] is a compelling approach to PMC, wherein state sets and the transition relation of the parameterized system are represented by finite automata. Standard fix point algorithms are not guaranteed to converge for this representation, hence more sophisticated techniques must be employed, such as *widening* [37].

Several papers have employed decidable logics to develop algorithms for PMC. Baukus et al. [14] use the *Weak Second Order Theory of One Successor*, while Fribourg and Olsén [58] and Maidl [92] employ *Presberger Arithmetic*.

5.5 Other Related Work

Issues surrounding the interplay between memory model and the compiler, hardware design, programmability, etc. are subtle and have received significant attention. For a general survey we refer the reader to the tutorial by Adve and Gharachorloo [4].

Several researchers in the multiprocessor community have looked at the problem of *dynamic verification* of memory model conformance. The idea here is to augment the multiprocessor memory system hardware with checker hardware that detects violations of the memory model. Two papers [25, 98] detect violations of SC arising from fabrication faults, transient faults, and design bugs, while another paper [64] detects violations of SC on a release consistent system arising from data races in the parallel code.

SC can be generalized to pertain to arbitrary concurrent objects (i.e. not just memories) being accessed by multiple clients. Another correctness condition for concurrent objects is *linearizability*, which was introduced by Herlihy and Wing [75]. When applied to shared memories, linearizability is stronger than SC but weaker than seriality. Unlike SC, linearizability is shown to be *local*, which means that a behavior is linearizable if and only if the behavior projected onto each individual object is linearizable. This is of course a desirable trait from a verification standpoint. For the case of shared memories, locality means that linearizability can be check on a *per-address basis*. From a computability perspective, linearizability is easier to handle than SC; Alur et al. [8] have shown linearizability to be decidable in exponential space.

Chapter 6

Conclusions and Future Work

This thesis has looked at two different goals related to formal verification of multiprocessor shared memory protocols, namely memory model verification and parameterized verification, with emphasis on model checking-based techniques.

Chapter 2 defined a subset of the memory model sequential consistency (SC), called *decisive sequential consistency* (DSC), that disallows read events to inherit values from writes that occur in their temporal futures. DSC can also be characterized as those SC traces that have serial reorderings that are also serial reorderings when restricted to any prefix of the trace. By colouring a prefix of a DSC trace black and the corresponding suffix white, and observing the number of black stripes in the reordering, we obtain a metric we called the shuffle degree. A trace is deemed DSC$_k$ if it has a DSC-reordering such that the shuffle degree never exceeds k over all prefixes. The chapter showed that checking DSC or DSC$_k$ for each $k \geq 2$ of a single trace is NP-complete. Also, the model checking problem for DSC$_k$ was shown to be decidable in EXPSPACE, while the problem for full DSC is PSPACE-hard; the decidability of the latter problem was left open. We also demonstrated that checking SC of our prefix-closed protocols is undecidable, refining a previous result of Alur et al. [8]. In support of the broadness of DSC, we argued that Lazy Caching, which is a well-known SC protocol that requires complex reorderings to witness, is DSC$_k$ for some k.

Chapter 3 tackled parameterized model checking (PMC) of SC. We presented an approach that uses an automatically generated abstraction to verify *simple SC* (SCC) of a family of protocols parameterized by the number of addresses and the number of data values per address. The number of processors is left fixed, however the number of addresses in the abstraction is dependent on

the number of processors. We defined a logic for describing protocols that supports the automatic generation of the abstraction. The abstraction always has the property that if it is SSC, then so too is the whole parameterized family. The approach was successfully applied to two interesting protocols. Finally, we contrasted our abstraction with a slightly different abstraction used by other authors, and showed how one might employ *noninterference lemmas* if the abstraction is too coarse.

Our final contribution was Chapter 4, where we looked at parameterization across the processor dimension. We verify properties that can be formulized as the unreachability of a set of error states. Since the technique is not specific to shared memory protocols, we refer to the components by the more general term *processes*. We define a class of systems called *nicely sliceable well-structured transition systems* (NSWs), which are a subclass of the well-studied *well-structured transition systems* (WSTSs). We showed that NSWs include petri nets and broadcast protocols (BP), which can model certain compositions of an arbitrary number of processes. BP are especially apt for modeling simple memory protocols. Roughly, our algorithm analyzes the system restricted to 1 process, then 2 processes, then 3, etc., until either a bug is found or a certain *convergence condition* is reached. Since each of these truncated systems is finite state, we can employ BDD-based symbolic model checking. We argued that when the constituent processes have large local state, our method can outperform the classical WSTS algorithm. This is confirmed by several experiments. We also defined a transformation that soundly reduces a system with *conjunctive guards* into a BP. The efficacy of our algorithm as along with the conjunctive guard reduction are demonstrated by successful verification of German's protocol with data paths.

The rest of this chapter presents several open questions arising from the work of this thesis, and, to various degrees, offers promising directions for their solution. Note that we designate statements as *conjectures* liberally; the conjectures we write in this chapter discuss notions that are at best defined only informally. Therefore, they are to be understood to have the implicit prefix *"There is some way to formalize all this stuff such that ..."*.

6.1 Decidability of DSC

As mentioned at the beginning of Sect. 2.6, the decidability of model checking DSC is an open problem. Of course, we have demonstrated its PSPACE-hardness, so resolving this question is of

theoretical interest only; even if it is decidable, the procedure will clearly be far from practical.[1]

As asserted by Theorem 11 on page 59, requiring prefix-closure of protocols leaves SC undecidable. The proof, however, exploits the ability of an SC protocol to "change its mind" about reorderings, which is disallowed by DSC. Hence there is no obvious means of adapting the proof of Theorem 11 to handle DSC. In an earlier paper of the author et al. [18], we conjectured that one could associate with any state of a DSC protocol a finite set of VWs that serve to summarize all traces that pass through the state. This would imply that every DSC protocol is DSC_k for some k. Assuming that the bound k was computable, this would imply decidability of DSC. Here we take a different approach, and suggest attempting to prove that an important subclass of DSC protocols are DSC_k.

As observed by Condon and Hu [36], real protocol descriptions involve a finite number L of *storage locations* that are variables having the type of data values. These storage locations are typically fields in cache entries, main memory, queue slots, or network messages. Data values are moved around the protocol by copying between storage locations. Let us call a protocol or trace *k-inheritance bounded* if at any point in time at most k write events occurring in the past can be inherited by read events occurring in the future.[2,3] Then clearly protocols with L storage locations are L-inheritance bounded. The following theorem relates the inheritance bound to the DSC bound for a very simple class of protocols.

Theorem 23. *Let $\mathcal{P}$ be a L-inheritance bounded DSC protocol involving 2 processors and 1 address. Then $\mathcal{P}$ is $\text{DSC}_{\mathcal{O}(L)}$.*

Proof. *(Sketch)* Let p_1 and p_2 be the two processors, let τ be a trace of $\mathcal{P}$, let π be a DSC-reordering of τ, let $\ell \in \mathbb{N}_{|\tau|}$, and let c_ℓ be as in Def. 3. The top part of Fig. 6.1 depicts the effect of the colouring c_ℓ on τ^π. Without loss of generality, one can prove that the rightmost black event of p_1 must occur in the leftmost black stripe, while the rightmost black event of p_2 must be the rightmost event of

[1] We emphasize that DSC is PSPACE-hard in the *size of the protocol* encoded as an automaton, which is already huge. So, in contrast to other PSPACE-hard problems which might have interesting yet small instances (for example, QBF), it is unlikely that any algorithm would be useful for any interesting examples.

[2] We are being a little sloppy here, since the inheritance relation is defined relative to a DSC-reordering (see page 12). What we really mean is that there exists a DSC-reordering π such that the inheritance relation induced by π is k-inheritance bounded.

[3] Condon and Hu [36] employ this property in their argument that certain protocols have *node bandwidth bounded (constraint) graphs*.

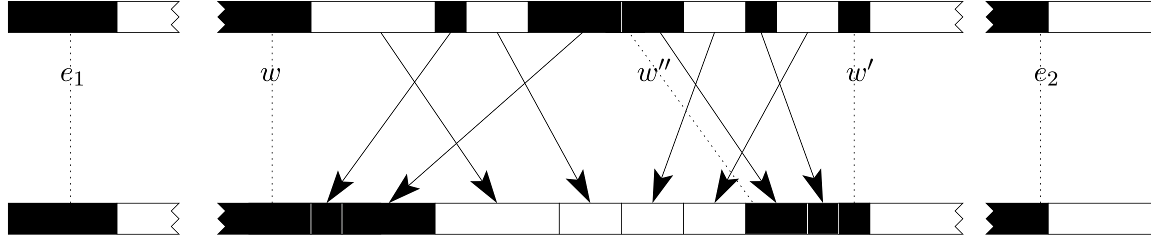


Figure 6.1: The stripe rearrangement described in the proof of Theorem 23

the rightmost black stripe. In Fig. 6.1, e_1 and e_2 respectively indicate the rightmost black events of processors p_1 and p_2. It follows that all black (resp. white) stripes except the leftmost (resp. rightmost) consist entirely of events of p_2 (resp. p_1). Since $\mathcal{P}$ is L-inheritance bounded, τ^π has at most L black writes that are inheritted by white reads. Let w and w' be consecutive black writes that are inheritted by white reads, and let w'' be the the first black write right of w (w'' is not necessarily distinct from w'). Fig. 6.1 depicts w, w' and w''. It can be shown that if all white stripes between w and w' are merged into a single white stripe, and this stripe is placed immediately prior to w'', the result is still a DSC-reordering. The bottom trace of Fig. 6.1 is an example of how this is done; the arrows show how stripes are shuffled and the dotted lines mark specific events . By iterating this stripe rearrangement we can reduce the number of black stripes, and therefore the shuffle degree, to be at most (approximately) L.

We have sketched how the shuffle degree of (τ, π, ℓ) can be reduced by using a new reordering π'. The DSC-bound is the maximum shuffle degree over all $i \in \mathbb{N}_{|\tau|}$. One can show that for any i, the shuffle degree of (τ, π', i) is not greater than (τ, π, i). Thus we can iterate our "reordering reordering" for all positions i that have shuffle degrees that are too high, yielding a final reordering with DSC-bound approximately equal to L. □

We believe that Theorem 23 can be easily generalized to an arbitrary number of processors. However, generalization to an arbitrary number of addresses will require a different proof technique, and probably require more assumptions. To illustrate the difficulty, consider the following trace.

$$\begin{gathered}(W, p_2, x, 2)\,[(W, p_2, y, 2)(R, p_2, x, 2)(W, p_2, x, 2)(R, p_2, y, 2)]^n \\ (R, p_1, x, 2)\,[(W, p_1, x, 1)(W, p_1, y, 1)]^n\,(R, p_1, x, 2)\end{gathered} \tag{6.1}$$

For any $n \geq 1$, the trace (6.1) can be shown to be DSC$_{2n}$ and not DSC$_{2n-1}$. However, with respect

to *any* DSC-reordering, (6.1) is a constant 3-inheritance bounded. The proof sketch of Theorem 23 argued that for single address traces, the inheritance bound and the DSC-bound (roughly) agree. The trace (6.1) demonstrates that this does not generalize for 2 or more addresses. However, it is possible that invoking data independence along with a pumping type argument, one could prove the following.

Conjecture 2. *If a protocol is DSC, data independent, and L-inheritance bounded, then it is DSC$_{\mathcal{O}(L)}$.*

6.2 Hardness of DSC$_k$

In Sect. 2.5 we showed that for any fixed $k \geq 1$, checking if a protocol is DSC$_k$ can be decided in exponential space. We have not explicitly provided any complexity theoretic lower bound for this problem. However, we note that since given any single trace τ, one can construct a protocol with trace set being precisely τ and its prefixes, a corollary of Theorem 5 is that for any $k \geq 2$, model checking DSC$_k$ is NP-hard. Here we suggest how our reduction of Sect. 2.6 (used to prove the PSPACE-hardness of full DSC) might be adapted to yield tighter lower bounds for DSC$_k$. A caveat: the following is very "hand wavy".

Recall that our reduction of Sect. 2.6 handles 3QBF formulae of the form (2.5), repeated here for convenience.

$$\forall x_n \exists y_n \cdots \forall x_1 \exists y_1 : f$$

We believe that our reduction can be adapted to handle the more general form wherein the first quantifier may be either $\forall$ or $\exists$, and there can be an arbitrary number of variables at each level of quantification (we still require that f is a 3CNF formula). Let us call such a formula a *generalized 3QBF* (G3QBF) formula.[4]

Hence, let us suppose that the reduction of Sect. 2.6 can be generalized to map any G3QBF formula φ to a protocol $\mathcal{P}_\varphi$ such that the notion of *intended trace* (see Def. 16) analogously generalizes; i.e. $\mathcal{P}_\varphi$ always has a trace it_φ such that Theorem 9 (which states that if it_φ is DSC then $\mathsf{evalq}(\varphi) = \mathsf{T}$) is preserved. Conversely, we envision that Theorem 10, which essentially states that

[4]Our notion of a G3QBF formula, in fact, coincides with the typical definition of a 3QBF formula; the terminology is simply meant to emphasize that we are generalizing the restricted form considered in Sect. 2.6

$\mathsf{evalq}(\varphi) = \mathsf{T}$ implies that $\mathcal{P}_\varphi$ is DSC, can be strengthened as follows, where $q(\varphi)$ is the number of quantifier alternations in φ.

Conjecture 3. *There exists some linearly bounded function $f : \mathbb{N} \to \mathbb{N}$ such that for any G3QBF formula φ we have $\mathsf{evalq}(\varphi) = \mathsf{T}$ implies $\mathcal{P}_\varphi \in DSC_{f(q(\varphi))}$.*

Assuming Conjecture 3, we find that given any G3QBF formula φ, we may construct a protocol $\mathcal{P}_\varphi$ such that

- if $\mathcal{P}_\varphi \in \mathrm{DSC}_{f(q(\varphi))}$, then $\mathsf{it}_\varphi \in$ DSC and hence by (the alleged generalization of) Theorem 9 we have $\mathsf{evalq}(\varphi) = \mathsf{T}$.
- if $\mathsf{evalq}(\varphi) = \mathsf{T}$ then $\mathcal{P}_\varphi \in \mathrm{DSC}_{f(q(\varphi))}$

In a mild abuse, let us define f^{-1} by $f^{-1}(k) = \max(\{n \mid f(n) \leq k\})$, and let $f_{\min}$ be the minimum value of f. We could therefore conclude that, for each $k \geq f_{\min}$, the problem of model checking DSC_k is hard for the $f^{-1}(k)th$ level of the polynomial time hierarchy [1, 47]. Analogously to our reduction for 3QBF in Sect. 2.6, the reduction we are proposing would use G3QBF restricted to ℓ quantifiers, which is complete for the ℓth level of the polynomial time hierarchy [47].

For the remainder of this section, we marshal evidence to support the feasibility of this reduction. To persuade the reader of Conjecture 3, we refer him or her to the proof of Theorem 10 on page 52, in particular the reordering (2.15). Fundamentally, (2.15) prescribes how *any* trace σ of $\mathcal{P}_\varphi$ is reordered; the underlined events are “older” events, which have a DSC reordering thanks to the inductive hypothesis of the proof, while the non-underlined events are the new events that are being shuffled into the reordering of the old. A key observation is that, if we let u be the length of the prefix of σ constituting the underlined events, then the shuffle degree (see Def. 3) of (σ, π, u) can easily be read off of (2.15): it is precisely the number of contiguous underlined subtraces, which is $1 + 2(n - k)$. Since $k \geq 1$, this shuffle degree is at most $2n - 1$. Note that the number of alternations in the 3QBF formula being considered in the proof of Theorem 10 is $2n$. Therefore, at least at the old/new interfaces of (2.15), the shuffle degree is *less than* the quantifier alternation.

There are two obvious outstanding issues hindering our furtherance of this proposal. First, we haven’t considered all the other interfaces obtained by underlining π' *and* some nonempty prefix of σ'', and observing the number of underlined stripes induced in (2.15). In fact this number is

not even well-defined, since in the proof of Theorem 10 we allow the traces CLS_0 and $\mathsf{CLS}(i)_1$ to arbitrarily order their constituent atomicity constructs. Clearly, if the DSC-bound of the reordering is to be curtailed, this ambiguity must be resolved. Likely, these traces must maintain the relative order of their atomicity constructs in CLS. Other components of (2.15) will need to be considered, and possibly reordered in a more "DSC-bound friendly" manner.

Second, we have not even *defined* how one reduces a G3QBF formula to a protocol, and once defined, the DSC-bound must also be shown to hold for this extended reduction. We believe that the reduction can be generalized, and the DSC-bound is likely preserved, modulo an additive or multiplicative constant. A key insight into how such a generalized reduction might work stems from a feature of the current reduction. Note that for formulas of the form (2.5), a pair of variables x_i and y_i "share" a heartbeat variable and all the surrounding widgets. These heartbeat variables allow for the unsatisfied clause widgets to have their reads satisfied, are responsible for coordinating the nested structure of the reorderings, and are responsible for the shuffling discussed above (one heartbeat variable contributes 2 to the shuffle degree). In the generalized reduction, it is probable that two entire groups of variables bound by consecutive universal and existential quantifiers can also share the same heartbeat address. Hopefully this can be arranged so that the additional variables at each level of quantification do not substantially increase the DSC-bound, and any increase is independent of the number of variables (which is unbounded).

6.3 Universal DSC

A primary result of this thesis is that model checking SC for our prefix-closed protocols is undecidable, and restricting attention to DSC does not appear amenable to (practical) algorithmic solution. Further, assuming that the hardness results hypothesized in Sect. 6.2 hold water, even considering the bounded DSC_k will be of little benefit. If we want to model check an interesting subset of SC then, we must adopt further real-world assumptions. One such assumption, formalized in Sezgin's work [117], is that of *universality*. An artifact of modeling protocols as general automata involving memory events is that we can define protocols that force events to happen with certain values. Of course, as observed by Sezgin, real protocols are slaves to the processors; if at some point in time a processor can write *some* value to an address, then it should be allowed to write *any* value to that address. Also, at any time there should be some computation path that allows a processor to read or

write any address. One way this property could be defined for our automata-based protocols is as follows.

Definition 54 (Strong Universality). *A protocol has* strong universality *if for any (reachable?) protocol state s, any processor p, and any address a, we have that for any data value d (resp. for some data value d), there exists a path starting from s such that the first memory event on the path is (W, p, a, d) (resp. (R, p, d, a)).*

Our experience with protocol descriptions suggests that Def. 54 might in fact be too strong. For example, in the Token Coherence protocol [93] when a processor has been granted a so-called *persistent request*, it will always perform at least one memory event to that address once it receives sufficient tokens. Hence, once the arbitration mechanism determines that a processor p is the "winner" for such a request, no other processor can perform a memory event to the address until *after* p does so. This suggests the following weaker notion of universality.

Definition 55 (Weak Universality). *A protocol has* weak universality *if for any (reachable?) protocol state s, any processor p, and any address a, we have that for any data value d (resp. for some data value d), there exists a path starting from s such that (W, p, a, d) (resp. (R, p, a, d)) eventually occurs on the path, and furthermore this is the first event by processor p on the path.*

Pinning down the "right" notion of universality, and exploring how this property can be both proved and exploited to assist in memory model verification could be an interesting and useful direction of research.

6.4 Extending/Improving Chapter 3

In Chapter 3 we presented a technique to do parameterized verification of a property called SSC, the parameters being addresses and data values. Aside from that fact that the number of processors is left fixed, we identify two weaknesses with this technique.

- Recall that our abstraction Q requires n concrete addresses, plus 1 abstracted address, where n is the number of processors. Hence the approach scales miserably with processors; adding a processor entails adding an address. Since most protocols have caches, which are modelled as two dimensional arrays indexed by processors and addresses, this has a dramatic impact

on the size of the state space. Ideally, the number of addresses in the abstraction would be a small constant, independent of the number of processors.

- SSC is quite a restricted version of SC; we would like to be able to handle protocols that are SC but not SSC, such as Lazy Caching.

Here we present the seed of an idea that, with some minor assistance from the user, could potentially attend to both of these concerns. Our idea handles the second weakness above, but with respect to only value-parameterized families. It is quite plausible that AVPPFs can also be handled, though here we do not make this argument. The method only involves observation of two addresses, regardless of the number of processors, therefore if it can be applied to AVPPFs, it will probably handle the first weakness also.

One way to characterize SC is as follows: a trace is SC if there exists a total order on the writes (that agrees with processor order) such that all processors observe (via their read events) this write order. Any SC protocol likely has a mechanism through which this intended write order is made explicit. If two write events of such a protocol are nondeterministically selected to have distinguished data values, it is possible that the description could be (manually) augmented so that a flag is set to indicate which of the two events should be ordered first. Presumably this augmentation, which we will call the *write order mechanism*, would involve non-interfering code that checks which of the two distinguished data values gets ordered first as they propagate through the protocol.

Assuming data independence, we may distinguish the two write events by considering the family member with the three data values $\{1, 2, 3\}$, and restricting the protocol so that at most one event writes value 2, and at most one event writes value 3. Code to perform this restriction can easily and automatically be inserted into the description. The write order mechanism is responsible for observing the protocol, and updating a fresh variable $order$. This variable has initial value U (meaning *unknown*), and is updated to 23 (resp. 32) once it is determined that the write with value 2 (resp. 3) should be reordered prior to the write with value 3 (resp. 2)

The write order mechanism is expected to have the following five properties, which we believe can be all be model checked; furthermore processor and address symmetry can likely be exploited to reduce the complexity of these checks.

Transitivity. For any three write events w_1, w_2, and w_3 occurring in a run, if the write order mech-

anism would order w_1 prior to w_2, and w_2 prior to w_3, then it must order w_1 prior to w_3.

Eventual order judgment. Any run in which both values 2 and 3 are written can be extended into a run where $order \neq U$; and this extension involves no memory events.

Respectfulness of processor order. If the writes with values 2 and 3 are performed by the same processor, then the write order mechanism updates *order* according to the temporal order in which writes appear.

Read order agreement. Let a_2 and a_3 respectively be the addresses that the data values 2 and 3 are written to. If $a_2 \neq a_3$ then along any run where eventually $order = 23$ (resp. 32), no processor may read 3 from a_3 (resp. read 2 from a_2) and then later read 1 from a_2 (resp. read 1 from a_3). Otherwise (i.e. $a_2 = a_3$) along any run where eventually $order = 23$ (resp. 32), no processor may read 3 (resp. 2) from a_3 and then later read 2 (resp. 3)

Order judgments are irrevocable. Once $order \neq U$, *order* must remain fixed forever.

For illustration, we return to our example of Lazy Caching. Referring to Fig. 2.3 on page 20, a write order mechanism satisfying all of the above can be realized by appending the following to the action for the event MemoryWrite$_i(a, d)$:

if $d \in \{2, 3\} \wedge order = U$ then
  $order :=$ (if $d = 2$ then 23 else 32) ;
endif

The following expresses our anticipation that these properties are sufficient to conclude SC.

Conjecture 4. *If a data independent value-parameterized protocol family restricted to* 3 *data values has a write order mechanism with the above five properties, then the family is SC.*

As demonstrated by the fact that a write order mechanism can be used to prove Lazy Caching SC, the approach can work for non-simple SC protocols. Thus, assuming the details can be successfully fleshed out, we have addressed our second concern regarding Chapter 3. Furthermore, noting that the approach only requires observation of two distinct addresses, if the approach can be shown to be compatible with our address abstraction then the first concern is addressed. In other words, it is our hope that the proposed approach of this section would enable one to perform address

parameterized verification of SC by model checking on an abstract protocol with 2 concrete and 1 abstract addresses, independent of the number of processors. Whether this can be done is unclear.

6.5 Future Directions for Chapter 4

For our work of Chapter 4, there are several obvious directions for further research. First, the current implementation is fairly naive. We believe more sophisticated symbolic model checking techniques might produce better results. We also require a thorough theoretical and empirical study of the behavior of our "lifting" operator on BDDs; a naive upper bound on the size of the BDD (4.19) on page 122 is $|\Gamma_{i-1}|^i$, where $|\Gamma_{i-1}|$ is the size of the BDD for Γ_{i-1}. Clearly, this bound does not yield good publicity for our approach.

Other directions include finding additional NSW applications, and finding ways to apply our method either semi-algorithmically or incompletely to systems that are not NSW. For example, in Sect. 4.4 we developed our so-called *conjunctive guard reduction*, which soundly eliminated a system trait that precluded classification as NSW. Another common trait in compositions of processes is having each process with one or more "pointers" to other processes. For example, the FLASH protocol [84], which has been used as an example in several papers on parameterized verification [28, 96, 105], has such state, as does the WIS protocol used in Sect. 3.6. Formally, these pointers are variables of type $P \to P$, where P is the type of processes. Unlike German's protocol, which has a single global process pointer, the process pointers of FLASH cannot be emulated by a variable of type $P \to \mathbb{B}$ and encoded in the finite process state. A sound reduction for state variables of the form $P \to P$ that allows our NSW approach of Chapter 4 to handle protocols such as FLASH would be a valuable contribution. Furthermore, state variables of type $P \times P \to \mathbb{B}$ also have a place in industrial protocols.

Bibliography

[1] S. Aaronson. The complexity zoo website. http://www.complexityzoo.com, 2005.

[2] P. A. Abdulla, K. Cerans, B. Jonsson, and T. Yih-Kuen. General decidability theorems for infinite-state systems. In *11th IEEE Symposium on Logic in Computer Science (LICS)*, pages 313–321, 1996.

[3] P. A. Abdulla and B. Jonsson. On the existence of network invariants for verifying parameterized systems. In *Correct System Design – Recent Insights and Advances, LNCS*, volume 1710, 1999.

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

[5] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):613–624, August 1993. Tech Report #1051, Univ. of Wisconsin.

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

[7] H. Akhiani, D. Doligez, P. Harter, L. Lamport, M. Tuttle, and Y. Yu. TLA+ verification of cache-coherence protocols. unpublished, available at http://research.microsoft.com/users/lamport/pubs/pubs.html#fm99, 1999.

[8] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. In *11th IEEE Symposium on Logic in Computer Science (LICS)*, 1996. Journal version is [9].

[9] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160(1-2):167–188, 2000.

[10] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 15:307–309, 1986.

[11] T. Arons. Using timestamping and history variables to verify sequential consistency. In *13th International Conference on Computer Aided Verification (CAV)*, pages 423–425, 2001.

[12] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *13th International Conference on Computer Aided Verification (CAV)*, pages 221–234, 2001.

[13] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *27th Annual International Symposium on Computer Architecture (ISCA)*, pages 282–293, 2000.

[14] K. Baukus, Y. Lakhnech, and K. Stahl. Verification of Parameterized Protocols. *Journal of Universal Computer Science*, 7(2):141–158, Feb 2001.

[15] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice/Hall, 1982.

[16] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast snooping: a new coherence method using a multicast address network. In *26th Annual International Symposium on Computer Architecture (ISCA)*, pages 294–304, 1999.

[17] J. Bingham. A new approach to upward closed set backward reachability analysis. In *6th International Workshop on Verification of Infinite-State Systems (INFINITY)*, 2004. Extended version: UBC Dept. of Computer Science Tech. Report TR-2004-07.

[18] J. Bingham, A. Condon, and A. J. Hu. Toward a decidable notion of sequential consistency. In *15th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 304–313, June 2003.

[19] J. Bingham, A. Condon, A. J. Hu, S. Qadeer, and Z. Zhang. Automatic verification of sequential consistency for unbounded addresses and data values. In *16th International Conference on Computer Aided Verification (CAV)*, pages 427–439, 2004.

[20] J. Bingham and A. J. Hu. Empirically efficient verification for a class of infinite-state systems. In *11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 77–92, 2005. Extended version: UBC Dept. of Computer Science Tech. Report TR-2005-07.

[21] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *12th International Conference on Computer Aided Verification (CAV)*, 2000.

[22] T. Braun, A. Condon, A. J. Hu, K. S. Juse, M. Laza, M. Leslie, and R. Sharma. Proving sequential consistency by model checking. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2001. Extended version: UBC Dept. of Computer Science Tech. Report TR-2001-03.

[23] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[24] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2), 1992.

[25] H. W. Cain and M. H. Lipasti. Verifying sequential consistency using vector clocks. In *14th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[26] J. F. Cantin, M. H. Lipasti, and J. E. Smith. The complexity of verifying memory coherence. In *15th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, June 2003.

[27] P. Chatterjee and G. Gopalakrishnan. A speciification and verification framework to design weak shared memory consistency protocols. In *4th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2002.

[28] C.T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2004.

[29] E. M. Clarke, O. Grumberg, and M. Browne. Reasoning about networks with many identical finite state processes. In *5th ACM Symposium on Principles of Distributed Computing*, pages 240–248, 1986.

[30] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *11th International Symposium on Computer Hardware Description Languages and their Applications*, 1993.

[31] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 19(5):726–750, 1997.

[32] W. W. Collier. Multiprocessor diagnostics website. http://www.mpdiag.com/.

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

[34] A. Condon, M. Hill, M. Plakal, and D. Sorin. Using lamport clocks to reason about relaxed memory models. In *5th International Symposium On High Performance Computer Architecture (HPCA)*, 1999.

[35] A. Condon and A. J. Hu. Automatable verification of sequential consistency. In *13th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 113–121, July 2001.

[36] A. Condon and A. J. Hu. Automatable verification of sequential consistency. *Theory of Computing Systems*, 36(5):431–460, 2003. Originally appeared as [35].

[37] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[38] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[39] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *11th International Conference on Computer Aided Verification (CAV)*, 1999.

[40] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *12th International Conference on Computer Aided Verification (CAV)*, July 2000.

[41] G. Delzanno, J. Esparza, and A. Podelski. Constraint-based analysis of broadcast protocols. In *Annual Conference of the European Association for Computer Science Logic*, pages 50–66, 1999.

[42] G. Delzanno and J. F. Raskin. Symbolic representation of upward-closed sets. In *6th International Conference Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 426–440, 2000.

[43] G. Delzanno, J. F. Raskin, and L. Van Begin. Attacking symbolic state explosion. In *13th International Conference on Computer Aided Verification (CAV)*, pages 298–310, 2001.

[44] L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with r distinct prime factors. *American Journal of Mathematics*, 35:413–422, 1913.

[45] D. Dill, S. Park, and A. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52, 1993.

[46] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[47] D.-Z. Du and K. Ko. *Theory of Computational Complexity*. John Wiley and Sons, 2000.

[48] Á. Th. Eiríksson and K. L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *7th International Conference on Computer Aided Verification (CAV)*, pages 367–380, 1995.

[49] E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *17th International Conference on Automated Deduction (CADE)*, pages 236–254, 2000.

[50] E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[51] E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *18th IEEE Symposium on Logic in Computer Science (LICS)*, pages 361–370, June 2003.

[52] E. A. Emerson and V. Kahlon. Rapid parameterized model checking of snoopy cache protocols. In *9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 144–159, 2003.

[53] E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *13th IEEE Symposium on Logic in Computer Science (LICS)*, pages 70–80, 1998.

[54] J. Esparza. Decidability and complexity of petri net problems – an introduction. In *Lectures on Petri Nets I: Basic Models. Advances in Petri Nets*, pages 374–428, 1998.

[55] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *14th IEEE Symposium on Logic in Computer Science (LICS)*, pages 352–359, 1999.

[56] A. Finkel. Reduction and covering of infinite reachability trees. *Information and Computation*, 89(2):144–179, 1990.

[57] A. Finkel and Ph. Schnoebelen. Well structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[58] L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into presburger arithmetic. In *8th International Conference on Concurrency Theory (CONCUR)*, pages 213–227, 1997.

[59] P. Ganty and L. Van Begin. Nondeterministic automata for the efficient verification of infinite-state. In *CP+CV Workshop at European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2004.

[60] D. Geer. Chip makers turn to multicore processors. *IEEE Computer*, 38(5):11–13, 2005.

[61] G. Geeraerts, J.-F. Raskin, and L. Van Begin. Expand, enlarge and check: new algorithms for the coverability problem of WSTS. In *24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 287–298, 2004.

[62] S. German. Personal correspondence. 2003.

[63] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, July 1992.

[64] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *3rd Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 1991.

[65] R. Ghughal and G. Gopalakrishnan. Verification methods for weaker shared memory consistency models. In *Workshop on Formal Methods for Parallel Programming*, pages 985–992, 2000. Workshop affiliated with the 14th International Parallel & Distributed Processing Symposium (IPDPS).

[66] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal of Computing*, 26(4):1208–1244, August 1997.

[67] M. Glusman and S. Katz. Extending memory consistency of finite prefixes to infinite computations. In *12th International Conference on Concurrency Theory (CONCUR)*, pages 411–425, 2001.

[68] G. Gopalakrishnan. A formalization of test model-checking, completeness results, and case studies. In *Workshop on Advances in Verification*, 2000. Workshop affiliated with the 12th International Conference on Computer Aided Verification (CAV).

[69] G. Gopalakrishnan, R. Ghughal, R. Hosabettu, A. Mokkedem, and R. Nalumasu. Formal Modeling and Validation Applied to a Commercial Coherent Bus: A Case Study. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 48–62, 1997.

[70] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *16th International Conference on Computer Aided Verification (CAV)*, pages 401–413, 2004.

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

[72] B. Heinemann. Subclasses of self-modifying nets. In *Applications and Theory of Petri Nets (Selected Papers from the First and Second European Workshop)*. Springer, 1982.

[73] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[74] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *11th International Conference on Computer Aided Verification (CAV)*, 1999.

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

[76] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society (3)*, 2(7):326–336, 1952.

[77] M. D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.

[78] A. J. Hu, M. Fujita, and C. Wilson. Formal verification of the HAL S1 system cache coherence protocol. In *International Conference on Computer Design*, pages 438–444. IEEE, 1997.

[79] C. N. Ip and D. L. Dill. Better verification through symmetry. In *International Conference on Computer Hardware Description Languages*, pages 87–100, April 1993.

[80] C. N. Ip and D. L. Dill. Verifying systems with replicated components in murphi. In *8th International Conference on Computer Aided Verification (CAV)*, 1996.

[81] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[82] Y. Kesten, A. Pnueli, E. Shahar, , and L. Zuck. Network invariants in action. In *13th International Conference on Concurrency Theory (CONCUR)*, 2002.

[83] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117(1), February 1995.

[84] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford FLASH multiprocessor. In *21st Annual International Symposium on Computer Architecture (ISCA)*, pages 302–313, 1994.

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

[86] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 267–281, 2004.

[87] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[89] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addision-Wesley, 2002.

[90] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and verifying systems with TLA+. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, pages 45–48, 2002.

[91] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized networks of processes. *Theoretical Computer Science*, 256:113–144, 2001.

[92] M. Maidl. A unifying model checking approach for safety properties of parameterized systems. In *13th International Conference on Computer Aided Verification (CAV)*, 2001.

[93] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Low-latency coherence on unordered interconnects. In *30th Annual International Symposium on Computer Architecture (ISCA)*, pages 182 – 193, June 2003.

[94] K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[95] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 219–234, 1999.

[96] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 179–195, 2001.

[97] K. L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In *International Symposium on Shared Memory Multiprocessing*, pages 242–251. Information Processing Society of Japan, 1991.

[98] A. Meixner and D. J. Sorin. Dynamic verification of sequential consistency. In *32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[99] M. Merritt. *Distributed Computing*, 12(2-3), 1999.

[100] R. Milner. An algebraic definition of simulation between programs. In *2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.

[101] M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall Inc., 1967.

[102] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), April 1989.

[103] R. Nalumasu. *Formal Design and Verification Methods for Shared Memory Systems*. PhD thesis, University of Utah, 1999.

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

[105] S. Park and D. Dill. Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. In *8th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 288–296, June 1996.

[106] S. Park and D. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, July 1998.

[107] S. Park and D. L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48(2):227–235, February 1999.

[108] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport clocks: Verifying a directory cache-coherence protocol. In *10th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998.

[109] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 82–97, 2001.

[110] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8), August 1995.

[111] F. Pong and M. Dubois. Formal automatic verification of cache coherence in multiprocessors with relaxed memory models. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):989–1006, September 2000.

[112] S. Qadeer. On the verification of memory models of shared-memory multiprocessors. In *Workshop on Formal Specification and Verification Methods for Shared Memory Systems*, October 2000. Workshop affiliated with 3th International Conference on Formal Methods in Computer-Aided Design (FMCAD).

[113] S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. Technical report, 2001. Compaq Systems Research Center Report 176, published later as [114].

[114] S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):730 – 741, August 2003.

[115] S. Qadeer and S. Rajamani. Personal correspondence. 2003.

[116] C. E. Scheurich. *Access ordering and coherence in shared memory multiprocessors*. PhD thesis, University of Southern California, 1989. Tech report CENG 89-19.

[117] A. Sezgin. *Formalization and Verification of Shared Memory*. PhD thesis, University of Utah, 2004.

[118] A. Sezgin and G. Gopalakrishnan. On the decidability of shared memory consistency verification. In *3rd ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2005.

[119] F. Somenzi. Colorado university decision diagram package (CUDD) webpage. http://vlsi.colorado.edu/∼fabio/CUDD/cuddIntro.html.

[120] D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. K. Martin, and D. A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6), June 2002.

[121] U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 206–224, 1995.

[122] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *5th ACM Symposium on Theory of Computing*, pages 1–9, 1973.

[123] J. E. Stoy, X. Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *Formal Methods Europe (FME)*, pages 43–71, 2001.

[124] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *13th ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–192, 1986.

[125] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, 1989.

[126] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[127] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 54–66, 1999.