National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Laboratory of Software Engineering

Optimal Exploration of Sequentially Consistent Executions of Concurrent Programs under View Equivalence

Diploma Thesis

by

Andreas Stamos

Supervisor: Konstantinos Sagonas,
Associate Professor NTUA

Athens, September 2025

THIS PAGE INTENTIONALLY LEFT BLANK.

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Βέλτιστη Εξερεύνηση Ακολουθιακά Συνεπών Εκτελέσεων Παράλληλων Προγραμμάτων υπό Ισοδυναμία Όψεων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ανδρέας Στάμος

ΕΠΙΒΛΕΠΩΝ: Κωνσταντίνος Σαγώνας,
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2025

THIS PAGE INTENTIONALLY LEFT BLANK.

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Laboratory of Software Engineering

Optimal Exploration of Sequentially Consistent Executions of Concurrent Programs under View Equivalence

Diploma Thesis

by

Andreas Stamos

Supervisor: Konstantinos Sagonas,
Associate Professor NTUA

Approved by the three-member examination committee

on 30 September 2025 in Athens, Greece

Konstantinos Sagonas	Nikolaos Papaspyrou	Zoi Paraskevopoulou
Associate Professor, ECE NTUA	Professor, ECE NTUA	Assistant Professor, ECE NTUA

Athens, September 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science
Laboratory of Software Engineering

Andreas Stamos

Graduate of School of Electrical and Computer Engineering,
National Technical University of Athens

To those who devoted their lives to establishing the foundations
and advancing the frontiers of modern computing.

Their vision enabled the limitless exploration of knowledge we experience today.

THIS PAGE INTENTIONALLY LEFT BLANK.

Περίληψη

Το σύγχρονο λογισμικό – από ελεγκτές αεροσκαφών έως εφαρμογές έξυπνων κινητών τηλεφώνων – εκτελείται σε διαρκώς και πιο παράλληλο υλικό. Ωστόσο, τα σφάλματα λόγω παραλληλίας παραμένουν κατά κοινή ομολογία δύσκολο να εντοπιστούν. Ένα παράλληλο πρόγραμμα μπορεί να εκτελεστεί με εκθετικά πολλούς διαφορετικούς τρόπους, ανάλογα με το πώς διαπλέκονται τα νήματά του στο χρονοπρόγραμμα. Εάν ακόμη και μία εκτέλεση προκαλεί σφάλμα, αυτή μπορεί τελικά να συμβεί, προκαλώντας δυνητικά καταστροφικές συνέπειες. Μια προσέγγιση για την επαλήθευση της απουσίας σφαλμάτων λόγω παραλληλίας σε ένα πρόγραμμα είναι το Stateless Model Checking (SMC), το οποίο εξερευνά συστηματικά τον χώρο των πιθανών εκτελέσεων (ή ένα αντιπροσωπευτικό υποσύνολο αυτών) εκτελώντας επανειλημμένα το πρόγραμμα από την αρχική κατάσταση έως τον τερματισμό υπό διαφορετικά σενάρια χρονοδρομολόγησης. Μια βασική ευκαιρία έγκειται στον πλεονασμό: πολλές εκτελέσεις επιδεικνύουν πανομοιότυπη συμπεριφορά του προγράμματος και επομένως δεν χρειάζεται να εκτελεστούν όλες. Ωστόσο, ακόμη και οι πιο εξελιγμένες παραλλαγές του SMC (βασισμένες σε ισοδυναμία happens-before, παρατηρήσιμης happens-before ή reads-from) εξακολουθούν να εξερευνούν πολλές φορές εκτελέσεις όπου όλα τα νήματα διαβάζουν τις ίδιες τιμές.

Αυτή η εργασία διερευνά την Ισοδυναμία Όψεων (View Equivalence), την πιο αδρή γνωστή σχέση ισοδυναμίας για το μοντέλο συνέπειας μνήμης της Ακολουθιακής Συνέπειας (Sequential Consistency): δύο εκτελέσεις χαρακτηρίζονται ισοδύναμες αν και μόνον αν κάθε ανάγνωση επιστρέφει την ίδια τιμή, ανεξάρτητα από το ποια εγγραφή την παρήγαγε. Εισαγάγουμε τον ViewXplore, τον πρώτο ορθό (sound) και βέλτιστο (optimal) αλγόριθμο για τη εξερεύνηση ακολουθιακά συνεπών εκτελέσεων παράλληλων προγραμμάτων υπό ισοδυναμία όψεων. Ο ViewXplore αναγνωρίζει όλες τις πιθανές τιμές κάθε ανάγνωσης και εξερευνά μία εκτέλεση ανά συνδυασμό τιμών, επιτυγχάνοντας να εκτελέσει τελικά ακριβώς μία εκτέλεση ανά κλάση ισοδυναμίας όψεων. Για να επιτύχει αυτό, ο ViewXplore απαιτεί έναν αλγόριθμο για την κατασκευή ακολουθιακά συνεπών εκτελέσεων. Εισαγάγουμε τον αλγόριθμο ConstructWitness, ο οποίος εκτελεί αυτή την κατασκευή σε πολυωνυμικό χρόνο για σχεδόν όλες τις εισόδους, παρά το γεγονός ότι το πρόβλημα αυτό είναι NP-πλήρες. Ο ViewXplore υλοποιείται για προγράμματα C/Pthreads μέσω ενσωμάτωσης με το εργαλείο Nidhugg που βασίζεται στο LLVM.

Επιπλέον, αποδεικνύουμε ότι το πρόβλημα απόφασης για το εάν οποιαδήποτε ακολουθιακά συνεπής εκτέλεση ενός δοθέντος παράλληλου προγράμματος παραβιάζει μια δοθείσα ιδιότητα, είναι NP-πλήρες. Αυτό καθιερώνει ένα κάτω φράγμα υπολογιστικής πολυπλοκότητας που ισχύει ανεξάρτητα από οποιαδήποτε επιλογή σχέσης ισοδυναμίας.

Λέξεις-κλειδιά: Παράλληλα Προγράμματα, Ακολουθιακή Συνέπεια, Επαλήθευση Προγραμμάτων, Stateless Model Checking, Ισοδυναμία Όψεων, ViewXplore, Κατασκευή Εκτελέσεων, NP-πληρότητα

THIS PAGE INTENTIONALLY LEFT BLANK.

Abstract

Modern software – from avionics controllers to smartphone apps – runs on increasingly parallel hardware. Yet concurrency bugs remain notoriously difficult to uncover. A concurrent program can execute in exponentially many different ways, depending on how its threads are interleaved. If even a single interleaving triggers a fault, it may eventually occur, potentially causing catastrophic consequences. A standard approach to verify the absence of concurrency errors in a program is *Stateless Model Checking (SMC)*, which systematically explores the space of possible interleavings (or a representative subset thereof) by repeatedly executing the program from the initial state to termination under different scheduling scenarios. A key opportunity lies in redundancy: many interleavings yield identical program behavior and therefore need not all be explored. However, even the most refined variants of SMC (based on happens-before, observers, or reads-from equivalence) still explore multiple times interleavings where all threads read the same values.

This thesis investigates *view equivalence*, the coarsest known equivalence relation for the Sequential Consistency (SC) memory model: two executions are equivalent if and only if every read returns the same value, irrespective of which write produced it. We introduce ViewXplore, the first *sound and optimal algorithm for exploring concurrent executions under view equivalence*. ViewXplore identifies all the possible values of each read and explores one relevant interleaving per combination of values, executing *exactly one* execution per view equivalence class. To achieve this, ViewXplore requires an algorithm to construct sequentially consistent executions. We introduce the ConstructWitness algorithm, which performs this construction in polynomial time for nearly all inputs, despite the problem being NP-complete. An *implementation* targeting C/Pthreads programs is built through integration with the LLVM-based Nidhugg stateless model checker.

In addition, we prove that the decision problem of *determining whether any SC execution violates a safety property* is NP-complete. This establishes a lower complexity bound that holds independently of any equivalence strategy.

Keywords: Concurrent Programs; Sequential Consistency; Program Verification; Stateless Model Checking; View Equivalence; ViewXplore; Execution Construction; NP-completeness

THIS PAGE INTENTIONALLY LEFT BLANK.

Contents

0 Εκτεταμένη Ελληνική Περίληψη (Extended Greek Abstract) 15

0.1 Εισαγωγή . . . 15
0.2 Συνεισφορές . . . 16
0.3 Θεωρητικό Υπόβαθρο . . . 16
0.4 Ισοδυναμία Όψεων . . . 18
0.5 Αλγόριθμος ViewXplore . . . 18
0.6 Αλγόριθμος ConstructWitness . . . 19
0.7 Μελλοντικές Επεκτάσεις . . . 20

1 Introduction 21

1.1 Relation to Previous Work . . . 22
1.2 Contributions . . . 23

2 Background 24

2.1 Notation . . . 24
2.2 The Sequential Consistency Model . . . 25
2.2.1 Key Relations of Sequential Consistency . . . 25
2.2.2 An Equivalent Graph-Based Characterisation of Sequential Consistency . . . 26
2.3 Stateless Model Checking . . . 27
2.4 Dynamic Partial Order Reduction . . . 29
2.4.1 Happens-Before Equivalence and the Optimal-DPOR Algorithm . . . 29
2.4.2 Observer Equivalence and the Optimal-DPOR-Observers Algorithm . . . 30
2.4.3 Reads-From Equivalence and the Optimal SMC-RF Algorithm . . . 30
2.4.4 Reads-Value-From (RVF) Equivalence and the SMC-RVF Algorithm . . . 31
2.4.5 Data-Centric Dynamic Partial Order Reduction . . . 31
2.5 Properties of SMC/DPOR Algorithms . . . 31
2.5.1 Soundness and Optimality . . . 31
2.5.2 Performance . . . 32
2.5.3 NP-completeness of Constructing Sequentially Consistent Executions . . . 33

3 View Equivalence 34

3.1 Definition . . . 34
3.2 Exponential Reduction in View Equivalence Classes . . . 35
3.2.1 The ManySameValue (msv) Program . . . 35
3.2.2 The ReadInc Program . . . 37

4 Exploration Algorithm 42

4.1 Definitions . . . 42
4.2 Algorithm . . . 43
4.3 Example: Running the Algorithm on ReadInc $[n = 2]$ . . . 46
4.4 Optimality . . . 47
4.5 Soundness . . . 48
4.6 Reducing Memory Consumption . . . 49
4.7 Improved Read-Cut Enumeration with Early Feasibility Pruning . . . 50

5 The Algorithm ConstructWitness **54**
5.1 ConstructWitnessComplete: A Complete Algorithm based on Constraint-Solving 54
5.2 Heuristic-URF: Fast Elimination of Inconsistent Executions 56
5.3 Heuristic-INCR: Incremental Construction of Sequentially Consistent Executions . . 59
5.4 Final ConstructWitness Algorithm 63

6 Why Naively Adapting the Reads-From Exploration Fails **64**

7 Implementation **66**
7.1 Forks and Joins 66
7.2 URF and INCR: Graph Representations 66
7.3 Nidhugg Integration 67
7.4 Flyweight Pattern 68
7.5 SMT Backend 68
7.6 An Optimized Membership Structure: The ContainsSet Data Structure 68

8 Evaluation **72**

9 MCSC: Model Checking for Sequential Consistency is NP-complete **73**
9.1 NP-hardness 73
9.2 On Execution Time of Programs 75
9.3 MCSC for polynomial-time programs is in NP 75

10 Future Work **76**

List of Definitions **77**

List of Theorems **77**

List of Algorithms **77**

References **79**

0 Εκτεταμένη Ελληνική Περίληψη (Extended Greek Abstract)

Η παρούσα ενότητα παρέχεται για κανονιστικούς λόγους και δεν εισάγει νέο περιεχόμενο ή συμπληρωματικές πληροφορίες.

This section is provided for regulatory purposes and does not introduce new content or supplementary information.

0.1 Εισαγωγή

Η εξάπλωση των πολυπύρηνων συστημάτων, απόρροια, μεταξύ άλλων, της κατάρρευσης της κλιμάκωσης του Dennard, έχει καταστήσει τον παράλληλο προγραμματισμό αναπόσπαστο μέρος σχεδόν κάθε εφαρμογής. Από κρίσιμα βιομηχανικά και ιατρικά συστήματα, μέχρι εφαρμογές κινητών τηλεφώνων, ο σωστός χειρισμός του παραλληλισμού συνιστά απαραίτητη προϋπόθεση για την ασφάλεια και την αξιοπιστία.

Ωστόσο, η ανάπτυξη παράλληλου λογισμικού κρύβει σημαντικές προκλήσεις. Σπάνιες και δύσκολα ανιχνεύσιμες αλληλεπιδράσεις μεταξύ νημάτων μπορούν να οδηγήσουν σε σφάλματα που εμφανίζονται απρόβλεπτα κατά τον χρόνο εκτέλεσης και είναι δύσκολο να αναπαραχθούν σε περιβάλλον δοκιμών. Σφάλματα όπως οι συνθήκες ανταγωνισμού (race conditions) ή οι παραβιάσεις ατομικότητας είναι χαρακτηριστικά παραδείγματα.

Για τον συστηματικό εντοπισμό τέτοιων σφαλμάτων έχει αναπτυχθεί η τεχνική *Stateless Model Checking (SMC)* [12, 10] (Κεφάλαιο 2.3), η οποία λαμβάνει ως είσοδο ένα παράλληλο πρόγραμμα και το εκτελεί επανειλημμένα με διαφορετικά χρονοπρογράμματα, δηλαδή διαφορετικές διατάξεις των νημάτων στο χρονοπρόγραμμα, ώστε να τρέξει ένα σύνολο εκτελέσεων που να καλύπτει όλες τις πιθανές συμπεριφορές που μπορεί να έχει το πρόγραμμα.

Αξίζει να διευκρινιστεί πως οι αλγόριθμοι Stateless Model Checking δεν έχουν πρόσβαση στον πηγαίο κώδικα του εξεταζόμενου προγράμματος, αλλά ελέγχουν μόνο τον χρονοδρομολογητή και λαμβάνουν γνώση μόνο για τις λειτουργίες που εκτελούνται στην κοινόχρηστη μνήμη (επηρεαζόμενη θέσης μνήμης και τιμή ανάγνωσης/εγγραφής ανά λειτουργία).

Μια πρωτόλεια ιδέα θα ήταν απλά να εκτελεστούν όλα τα πιθανά χρονοπρογράμματα, δηλαδή όλες οι πιθανές διατάξεις νημάτων. Το βασικό πρόβλημα αυτής της ιδέας έγκειται στο γεγονός ότι ένα πολυνηματικό πρόγραμμα μπορεί να εκτελεστεί με εκθετικά πολλά διαφορετικά χρονοπρογράμματα, καθώς οι πιθανές διατάξεις των νημάτων είναι εκθετικά πολλές. Αποδεικνύεται σε αυτή την εργασία (Θεώρημα 2.2) πως ένα πρόγραμμα με k νήματα και n λειτουργίες ανά νήμα επιδέχεται $\frac{(kn)!}{(n!)^k}$ διαφορετικά χρονοπρογράμματα. Ενδεικτικά για $k = 3$ νήματα και $n = 10$ λειτουργίες ανά νήμα υπάρχουν $5.5 \cdot 10^{12}$ (5.5 τετράκις-εκατομμύρια) χρονοπρογράμματα. Ακόμη κι αν ένα μόνο από τα εκθετικά πολλά πιθανά χρονοπρογράμματα παραβιάζει κάποια δοσμένη ιδιότητα ασφαλείας, το πρόγραμμα θεωρείται μη ασφαλές, και το χρονοπρόγραμμα αυτό πρέπει να εντοπιστεί.

Προκειμένου να αποφευχθεί η εξαντλητική αναζήτηση όλων των πιθανών χρονοπρογραμμάτων, τα τελευταία χρόνια έχει δοθεί έμφαση στην αναζήτηση αλγορίθμων που αποφεύγουν τον έλεγχο ισοδύναμων εκτελέσεων (Κεφάλαιο 2.4). Οι αλγόριθμοι αυτοί διαμοιράζουν όλες τις πιθανές εκτελέσεις σε κλάσεις ισοδυναμίας με βάση μια καθορισμένη σχέση ισοδυναμίας και έπειτα επιχειρούν να εκτελέσουν τουλάχιστον μια εκτέλεση, ή ιδανικότερα ακριβώς μία εκτέλεση, από κάθε κλάση ισοδυναμίας.

Ο χαρακτηρισμός της ισοδυναμίας διαφέρει από αλγόριθμο σε αλγόριθμο, όμως πάντα δύο εκτελέσεις

που χαρακτηρίζονται ισοδύναμες εμφανίζουν την ίδια συμπεριφορά. Πιο συγκεκριμένα, καθώς τα νήματα θεωρούνται αιτιοκρατικά δοθέντων των τιμών που διαβάζουν από την κοινόχρηστη μνήμη, για να χαρακτηριστούν δύο εκτελέσεις ισοδύναμες πρέπει όλα τα νήματα να έχουν διαβάσει σε όλες τις λειτουργίες ανάγνωσης ίδιες τιμές από την κοινόχρηστη μνήμη. Ωστόσο, δύο εκτελέσεις μπορεί να έχουν ίδιες τιμές ανάγνωσης σε όλες τις λειτουργίες ανάγνωσης (οπότε ίδια συμπεριφορά) και παρ' όλα αυτά μια αποδεκτή σχέση ισοδυναμίας να μην τις χαρακτηρίσει ισοδύναμες. Μια τέτοια σχέση ισοδυναμίας θα αναγκάσει έναν αλγόριθμο που στηρίζεται σε αυτή να εξερευνήσει πάνω μία εκτέλεση ανά συμπεριφορά προγράμματος.

Η εργασία αυτή μελετά μία νέα σχέση ισοδυναμίας εκτελέσεων, την Ισοδυναμία Όψεων (Κεφάλαιο 3 και ορισμός 3.1), σύμφωνα με την οποία, ο παραπάνω πλεονασμός εξαλείφεται και πλέον δύο εκτελέσεις χαρακτηρίζονται ισοδύναμες αν και μόνο αν όλα τα νήματα διαβάζουν ίδιες τιμές για όλες τις λειτουργίες ανάγνωσης. Η εργασία αυτή εισαγάγει τον ViewXplore, τον πρώτο αλγόριθμο που εξερευνά ακριβώς μια εκτέλεση ανά κλάση Ισοδυναμίας Όψεων. Ακριβώς επειδή ο πλεονασμός των παλαιότερων σχέσεων ισοδυναμίας εξαλείφεται, το πλήθος των κλάσεων ισοδυναμίας μειώνεται και έτσι ο χρόνος εκτέλεσης του αλγορίθμου εξερεύνησης μειώνεται και αυτός.

0.2 Συνεισφορές

Η παρούσα εργασία περιλαμβάνει τέσσερις συνεισφορές, οι οποίες από κοινού εξελίσσουν τη θεωρία και την πρακτική της επαλήθευσης παράλληλων προγραμμάτων. Οι συνεισφορές αυτές είναι:

1. Εισαγάγεται ο αλγόριθμος ViewXplore, ο πρώτος αλγόριθμος εξερεύνησης εκτελέσεων παράλληλων προγραμμάτων υπό ισοδυναμία όψεων για το μοντέλο της Ακολουθιακής Συνέπειας (Sequential Consistency) μνήμης. Ο αλγόριθμος αυτός αποδεικνύεται ορθός (sound) και βέλτιστος (optimal), δηλαδή εκτελεί *ακριβώς μία* αντιπροσωπευτική εκτέλεση ανά κλάση ισοδυναμίας όψεων (Κεφάλαιο 4 και θεωρήματα 4.1 και 4.2).

2. Εισαγάγεται αλγόριθμος για το NP-πλήρες πρόβλημα της κατασκευής Ακολουθιακά Συνεπών (Sequentially Consistent) εκτελέσεων. Ο αλγόριθμος αποτελείται από δύο ευριστικά στάδια με πολυπλοκότητα πολυωνυμικού χρόνου, τα οποία βασίζονται σε γράφους, και ένα τελικό στάδιο που βασίζεται σε επίλυση προβλήματος περιορισμών (Κεφάλαιο 5). Τα ευριστικά στάδια επιλύουν σε πολυωνυμικό χρόνο σχεδόν το σύνολο των πιθανών εισόδων.

3. Ο ViewXplore υλοποιείται και ενσωματώνεται στο εργαλείο Nidhugg, που εκτελεί προγράμματα LLVM IR, προσφέροντας πλήρη υποστήριξη για προγράμματα C/Pthreads (Κεφάλαιο 7).

4. Αποδεικνύεται ότι το γενικότερο πρόβλημα απόφασης της ύπαρξης Ακολουθιακά Συνεπούς εκτέλεσης που παραβιάζει μια δοσμένη ιδιότητα (MCSC) είναι NP-πλήρες, κάτι που καθορίζει τα θεωρητικά όρια κάθε μεθόδου ελέγχου (Κεφάλαιο 9 και θεωρήματα 9.1 και 9.2). Το αποτέλεσμα αυτό δεν εντοπίζεται ως σήμερα στην βιβλιογραφία.

0.3 Θεωρητικό Υπόβαθρο

Η κατανόηση της συμπεριφοράς των παράλληλων προγραμμάτων βασίζεται στην έννοια της εκτέλεσης και στη σχέση μεταξύ των ενεργειών των νημάτων, δηλαδή στο λεγόμενο μοντέλο συνέπειας (memory consistency model). Κάθε εκτέλεση μπορεί να αναπαρασταθεί ως ένα σύνολο λειτουργιών *ανάγνωσης (read)* και *εγγραφής (write)* πάνω σε κοινόχρηστες θέσεις μνήμης.

Στο μοντέλο **Ακολουθιακής Συνέπειας (Sequential Consistency, SC)** [20] (Κεφάλαιο 2.2), που εισήγαγε ο Lamport, απαιτείται να υπάρχει μία συνολική, σειριακή διάταξη όλων των ενεργειών, η οποία:

1. Διατηρεί τη σειρά των ενεργειών κάθε νήματος, και
2. Αποδίδει σε κάθε ανάγνωση την τιμή της πιο πρόσφατης, σύμφωνα με την σειριακή διάταξη, εγγραφής στην ίδια διεύθυνση μνήμης.

Με άλλα λόγια, η SC μοντελοποιεί έναν επεξεργαστή, όπου τα νήματα εκτελούνται παράλληλα μεν, αλλά το αποτέλεσμα είναι σαν να είχαν εκτελεστεί σειριακά.

Στη βιβλιογραφία έχουν προταθεί διάφορες σχέσεις ισοδυναμίας εκτελέσεων (Κεφάλαιο 2.4). Οι σημαντικότερες είναι οι εξής:

1. **Ισοδυναμία Happens-Before [1, 3] (Κεφάλαιο 2.4.1):** Εισάγει τη σχέση happens-before, η οποία εκφράζει την αιτιότητα μεταξύ ενεργειών. Πιο συγκεκριμένα μια λειτουργία είναι happens-before από μια δεύτερη λειτουργία αν οι δύο λειτουργίες αφορούν την ίδια διεύθυνση μνήμης και τουλάχιστον μία από τις δύο είναι λειτουργία εγγραφής.

   Δύο εκτελέσεις χαρακτηρίζονται ισοδύναμες αν επάγουν την ίδια σχέση happens-before.

2. **Ισοδυναμία Παρατηρήσιμης (Observable) Happens-Before [7] (Κεφάλαιο 2.4.2):** Εισάγει τη σχέση παρατηρήσιμης happens-before. Πιο συγκεκριμένα μια λειτουργία εγγραφής είναι παρατηρήσιμα happens-before από μια λειτουργία ανάγνωσης πάντα, αν οι δύο λειτουργίες αφορούν την ίδια διεύθυνση μνήμης. Ωστόσο, μια λειτουργία εγγραφής είναι παρατηρήσιμα happens-before από μια λειτουργίας εγγραφής αν και μόνο αν υπάρχει μια λειτουργία ανάγνωσης που διαβάζει από την δεύτερη λειτουργία εγγραφής. Σημειώνεται πως θεωρούμε ότι μια λειτουργία ανάγνωσης διαβάζει από μια λειτουργία εγγραφής όταν η λειτουργία εγγραφής είναι η τελευταία λειτουργία εγγραφής πριν την λειτουργία ανάγνωσης για την διεύθυνση μνήμης που διαβάζει η λειτουργία ανάγνωσης. Δύο εκτελέσεις χαρακτηρίζονται ισοδύναμες αν επάγουν την ίδια σχέση παρατηρήσιμης happens-before.

3. **Ισοδυναμία Πηγής Ανάγνωσης [5] (Κεφάλαιο 2.4.3):** Δύο εκτελέσεις είναι ισοδύναμες αν όλες οι λειτουργίες ανάγνωσης των δύο εκτελέσεων διαβάζουν από την ίδια λειτουργία εγγραφής στις δύο εκτελέσεις. Σημειώνεται, όπως και πριν, πως θεωρούμε ότι μια λειτουργία ανάγνωσης διαβάζει από μια λειτουργία εγγραφής όταν η λειτουργία εγγραφής είναι η τελευταία λειτουργία εγγραφής πριν την λειτουργία ανάγνωσης για την διεύθυνση μνήμης που διαβάζει η λειτουργία ανάγνωσης.

Και για τις τρεις παραπάνω σχέσεις ισοδυναμίας έχουν προταθεί ορθοί και βέλτιστοι αλγόριθμοι, δηλαδή αλγόριθμοι που εξερευνούν ακριβώς μια εκτέλεση ανά κλάση ισοδυναμίας.

Παρότι οι παραπάνω σχέσεις ισοδυναμίας επάγουν σημαντικά μικρότερο πλήθος κλάσεων ισοδυναμίας σε σχέση με το πλήθος των εφικτών χρονοπρογραμμάτων, παραμένουν μη βέλτιστες. Ειδικότερα, συχνά θεωρούν δύο εκτελέσεις μη ισοδύναμες παρόλο που οδηγούν στις ίδιες τιμές ανάγνωσης και άρα στην ίδια συμπεριφορά προγράμματος.

Για παράδειγμα, αν δύο εγγραφές γράφουν και οι δύο την τιμή 0 και μια ανάγνωση τη διαβάζει, το πρόγραμμα δεν μπορεί να διακρίνει ποια από τις δύο εγγραφές παρήγαγε τη συγκεκριμένη τιμή που διάβασε η λειτουργία ανάγνωσης. Ωστόσο, και οι τρεις παραπάνω σχέσεις ισοδυναμίας θα χαρακτηρίσουν δύο

εκτελέσεις στις οποίες η λειτουργία ανάγνωσης διάβασε από διαφορετική λειτουργία εγγραφής την ίδια τιμή, ως μη ισοδύναμες.

0.4 Ισοδυναμία Όψεων

Αυτό ακριβώς το κενό καλύπτει η **Ισοδυναμία Όψεων (View Equivalence)**, που μελετάται στην παρούσα εργασία (Κεφάλαιο 3), η οποία ορίζει την ισοδυναμία με βάση μόνο τις τιμές που τελικά διαβάζονται, ανεξάρτητα από την προέλευσή τους (Ορισμός 3.1).

Δύο εκτελέσεις είναι ισοδύναμες αν όλες οι λειτουργίες ανάγνωσης διαβάζουν τις ίδιες τιμές στις δύο εκτελέσεις.

Η σχέση ισοδυναμίας αυτή μειώνει σημαντικά το πλήθος κλάσεων ισοδυναμίας, επιτρέποντας συνακόλουθα την μείωση του χρόνου εκτέλεσης για την εξερεύνηση των εκτελέσεων ενός παράλληλου προγράμματος (θεωρώντας αλγορίθμους που εξερευνούν ακριβώς μία εκτέλεση ανά κλάση ισοδυναμίας).

Παρουσιάζονται δύο προγράμματα-παραδείγματα που δείχνουν την ισχύ της Ισοδυναμίας Όψεων στην εκθετική μείωση των κλάσεων ισοδυναμίας:

1. ManySameValue (msv) (Κεφάλαιο 3.2.1)

   Το πρόγραμμα αποτελείται από δύο νήματα: το πρώτο γράφει επανειλημμένα την τιμή 0 σε μια κοινόχρηστη μεταβλητή, ενώ το δεύτερο τη διαβάζει. Αποδεικνύεται στην παρούσα εργασία ότι, με βάση τις παλαιότερες σχέσεις ισοδυναμίας, το πλήθος των εκτελέσεων και των αντίστοιχων κλάσεων ισοδυναμίας αυξάνεται ασυμπτωτικά εκθετικά ως προς το πλήθος των λειτουργιών (Θεωρήματα 3.1 και 3.2 και πόρισμα 3.1). Ωστόσο, σε όλες τις εκτελέσεις όλες οι λειτουργίες ανάγνωσης διαβάζουν την τιμή 0. Έτσι, το πρόγραμμα έχει ακριβώς μία κλάση Ισοδυναμίας Όψεων.

2. ReadInc (Κεφάλαιο 3.2.2)

   Κάθε νήμα διαβάζει μια κοινόχρηστη μεταβλητή `x` και στη συνέχεια γράφει την τιμή `x + 1`. Μετράται ότι ο αριθμός των διαφορετικών κλάσεων ισοδυναμίας αυξάνεται πολύ πιο αργά με την αύξηση των νημάτων, ως προς την Ισοδυναμία Όψεων σε σχέση με τις άλλες σχέσεις ισοδυναμίας (Πίνακας 1 και σχήμα 2).

0.5 Αλγόριθμος ViewXplore

Ο ViewXplore είναι ο πρώτος αλγόριθμος που εξερευνά ακριβώς μία εκτέλεση για κάθε κλάση Ισοδυναμίας Όψεων (Κεφάλαιο 4). Για να το επιτύχει, εισάγει την έννοια του *read-cut* (Ορισμός 4.1), δηλαδή τον συνδυασμό, για όλα τα νήματα, προθεμάτων μιας ακολουθίας τιμών που διαβάζει κάθε νήμα.

Σε αδρές γραμμές ο αλγόριθμος ακολουθεί τα εξής βήματα (Αλγόριθμος 1):

1. Τρέχει μια εκτέλεση.
2. Διατρέχει όλα τα read-cuts της εκτέλεσης:
   i. Για κάθε νήμα, επεκτείνει κάθε read-cut με μια λειτουργία ανάγνωσης που διαβάζει διαφορετική τιμή από την τρέχουσα εκτέλεση. Γίνονται επεκτάσεις για όλες τις διαφορετικές τιμές που εγγράφονται στην συγκεκριμένη διεύθυνση μνήμης στην συγκεκριμένη εκτέλεση[1]. Αν το επεκτεταμένο read-cut ανήκει σε εκτέλεση που εξερευνήθηκε στο παρελθόν αγνοείται.

[1] ακριβέστερα για όλες τις τιμές που εγγράφονται από εγγραφές που επιτρέπει το συγκεκριμένο read-cut.

ii. Με τον αλγόριθμο κατασκευής ακολουθιακά συνεπών εκτελέσεων ConstructWitness, που αυτή η εργασία εισαγάγει, επιχειρεί να κατασκευάσει ένα πρόθεμα εκτέλεσης, αν υπάρχει, που να επάγει το επεκτεταμένο read-cut. Αν δεν υπάρχει, το επεκτεταμένο αυτό read-cut αγνοείται.

iii. Αν προηγουμένως κατασκευάστηκε πρόθεμα εκτέλεσης, τρέχει μια νέα πλήρη εκτέλεση για το πρόθεμα αυτό χρησιμοποιώντας τον χρονοδρομολογητή. Τοποθετεί την εκτέλεση σε μια δομή δεδομένων που διατηρεί εκτελέσεις προς ανάλυση. Για λόγους μείωσης της χρήσης μνήμης, όπως εξηγείται, επιλέγεται ως δομή δεδομένων μια ουρά, αν και η επιλογή αυτή δεν επηρεάζει την ορθότητα της εξερεύνησης.

3. Αφού επαναλάβει την παραπάνω διαδικασία για όλα τα read-cuts της εκτέλεσης που είχε τρέξει, προχωράει, όπως παραπάνω, στην ανάλυση της επόμενης εκτέλεσης που υπάρχει στην δομή δεδομένων που διατηρεί τις εκτελέσεις προς ανάλυση.

4. Όταν δεν απομένουν άλλες εκτελέσεις προς ανάλυση, ο αλγόριθμος εξερεύνησης τερματίζει.

Επισημαίνεται ότι το παραπάνω συνιστά μονάχα μια αδρή, υπεραπλουστευμένη περιγραφή του αλγόριθμου, που απέχει σημαντικά από τον πλήρη αλγόριθμο ο οποίος αναλύεται παρακάτω.

Αποδεικνύεται ότι ο ViewXplore είναι:

1. **Ορθός:** Εξερευνά τουλάχιστον μια εκτέλεση από κάθε κλάση Ισοδυναμίας Όψεων. (Θεώρημα 4.2)
2. **Βέλτιστος:** Εξερευνά ακριβώς μια εκτέλεση από κάθε κλάση Ισοδυναμίας Όψεων. (Θεώρημα 4.1)

0.6 Αλγόριθμος ConstructWitness

Η κατασκευή Ακολουθιακά Συνεπών (SC) εκτελέσεων για δοθείσες ακολουθίες λειτουργιών ανά νήμα έχει αποδειχθεί πως είναι ένα NP-πλήρες πρόβλημα, δηλαδή ένα υπολογιστικά δύσκολο πρόβλημα. Πιο συγκεκριμένα, το πρόβλημα απόφασης αν υπάρχει Ακολουθιακά Συνεπής εκτέλεση είναι NP-πλήρες [11].

Παρ' όλα αυτά, εμείς εισαγάγουμε τον αλγόριθμο ConstructWitness που κατασκευάζει σε πολυωνυμικό χρόνο Ακολουθιακά Συνεπείς εκτελέσεις σχεδόν για όλες τις πιθανές εισόδους (Κεφάλαιο 5). Πιο συγκεκριμένα, ο ConstructWitness χρησιμοποιεί τους ακόλουθους τρεις μηχανισμούς:

1. Heuristic-URF (Κεφάλαιο 5.2): Στόχος του πρώτου σταδίου είναι να δείξει σε πολυωνυμικό χρόνο ότι για κάποιες εισόδους δεν υπάρχει ακολουθιακά συνεπής εκτέλεση. Αυτό επιτυγχάνεται κατασκευάζοντας σταδιακά μια αναγκαστική σχέση happens-before, υπό την έννοια πως αν υπάρχει-κάποια ακολουθιακά συνεπής εκτέλεση, τότε αυτή θα ικανοποιεί την αναγκαστική happens-before σχέση. Η σχέση αυτή πρέπει να είναι ακυκλική. Αν σε κάποιο σημείο, εισαγάγουμε κύκλο τότε είναι αδύνατο να υπάρχει ακολουθιακά συνεπής εκτέλεση.

   Το στάδιο αυτό επιστρέφει είτε *ασυνεπές* ή *άγνωστο*.

2. Heuristic-INCR (Κεφάλαιο 5.3): Στόχος του δεύτερου σταδίου εί ναι να κατασκευάσει μια εκτέλεση, χρησιμοποιώντας τόσο την αναγκαστική σχέση happens-before που υπολογίστηκε προηγουμένως, όσο και την διάταξη των λειτουργιών της εκτέλεσης από την οποία παράχθηκε η είσοδος του ConstructWitness στον ViewXplore. Πιο συγκεκριμένα, παρατηρείται ότι στον ViewXplore όλες οι είσοδοι της ConstructWitness προκύπτουν με "μικρές μεταβολές" από

μια αρχική ακολουθιακά συνεπή εκτέλεση. Έτσι, είναι πιθανό, χρησιμοποιώντας την διάταξη των λειτουργιών αυτής να μπορέσουμε να κατασκευάσουμε μια ακολουθιακά συνεπή εκτέλεση.

Στην πραγματικότητα, η κατασκευή είναι σημαντικά πιο σύνθετη από αυτό που περιγράφεται εδώ, και αυτό που γίνεται είναι πως κατασκευάζεται μια *επαρκής* happens-before (προσδιορίζεται τι σημαίνει το *επαρκής*) και αν προκύψει ακυκλική, επιστρέφουμε ως εκτέλεση μια τοπολογική της ταξινόμηση, ενώ αν προκύψει κυκλική επιστρέφουμε *άγνωστο*.

Το στάδιο αυτό επιστρέφει είτε ένα *πρόθεμα εκτέλεσης* ή *άγνωστο*.

3. ConstructWitnessComplete (Κεφάλαιο 5.1) Αν και τα δύο παραπάνω στάδια επιστρέψουν *άγνωστο* καταφεύγουμε σε ένα τρίτο στάδιο που αποφαίνεται για όλες τις εισόδους είτε *ασυνεπές* είτε με *πρόθεμα εκτέλεσης*. Ο αλγόριθμος αυτός βασίζεται σε αναγωγή του προβλήματος σε επίλυση προβλήματος περιορισμών.

Στην πράξη, οι δύο πρώτες τεχνικές επιτυγχάνουν σχεδόν πάντα, εξασφαλίζοντας υψηλή επίδοση (πολυωνυμικό χρόνο).

Σημειώνεται ότι με δεδομένο ότι έχει αποδειχθεί ότι το πρόβλημα κατασκευής ακολουθιακά συνεπών εκτελέσεων είναι NP-πλήρες [11], εκτός αν P = NP, δεν υπάρχει αλγόριθμος πολυωνυμικού χρόνου που να αποφαίνεται για όλες τις εισόδους. Αυτό φυσικά δεν αποκλείει την παραπάνω κατασκευή, όπου πολυωνυμικό χρόνο απαιτούν οι περισσότερες είσοδοι αλλά όχι όλες.

0.7 Μελλοντικές Επεκτάσεις

Μελλοντικά, η εργασία αυτή μπορεί να επεκταθεί (Κεφάλαιο 10):

1. Με την υποστήριξη ατομικών λειτουργιών Read-Modify-Write και κλειδωμάτων.
2. Με την υποστήριξη πιο χαλαρών μοντέλων συνέπειας μνήμης (όπως Total Store Order, Partial Store Order και Release/Acquire).
3. Με την παραλληλοποίηση του αλγορίθμου εξερεύνησης για αύξηση επίδοσης.

1 Introduction

Writing correct concurrent software is hard. Race conditions, atomicity violations, and misplaced memory fences manifest rarely, depend on timing, and may survive months or even years of field testing – sometimes only being discovered when they cause catastrophic failures. Notorious incidents, e.g., the Mars Pathfinder resets [14], the 2003 Northeast Blackout [22], and the Therac-25 radiation overdoses [21], can all be traced back to subtle race conditions and other concurrency errors.

Underlying the above failures is the fact that the execution of a concurrent program is inherently non-deterministic due to scheduling non-determinism. A concurrent program may be executed under multiple possible schedules, each exhibiting potentially different behaviors depending on the order in which the scheduler interleaves the operations of the participating threads.

Regardless of the scheduling decisions, a correct concurrent program must preserve its intended semantics. Ensuring this property is the responsibility of the programmer.

A natural need that arises from this challenge is for a tool that can execute all feasible interleavings that a concurrent program may exhibit. Such a tool would allow a programmer or verifier to explicitly check whether certain properties hold in all feasible executions - for instance, that the program never crashes or violates an assertion. However, the number of possible interleavings of a concurrent program grows exponentially with the number of threads and shared accesses. For a program with k threads and n operations per thread, there exist $\frac{(kn)!}{(n!)^k}$ distinct interleavings. For example, with $k = 3$ and $n = 10$, the number of possible interleavings is $5.5 \cdot 10^{12}$. Consequently, exhaustively executing all feasible interleavings is, in general, computationally infeasible.

Fortunately, many interleavings result in the same program behavior. In particular, consider two different interleavings in which all threads read exactly the same values at every read operation. Since the reads read identical values, both interleavings induce the same thread-local behaviors, and therefore the same global program behavior. This observation motivates the idea of reducing exploration by pruning equivalent interleavings, keeping only one representative execution per distinct combination of read values. In essence, the goal is to explore exactly one execution for each distinct combination of values read by all threads.

Example 1.1. As a first example, consider a program with two concurrent threads, p and q, assigning the same value to a shared variable `x`, initially set to `0`.

p	q
`x := 1`	`x := 1;`
	`a := x`

This simple program has three possible interleavings ($p_1q_1q_2$, $q_2p_1q_2$, and $q_1q_2p_1$, where with t_i we denote the i-th operation of thread t). In all interleavings, the operation q_2, which reads `x` and assigns its value to a local register `a`, reads the value `1`. Consequently, all interleavings are behaviorally equivalent, and it suffices to execute only one of them.

A key challenge is that the set of possible values a read may read is not known a priori in programs. Some writes that could be read from may only appear after certain threads perform reads with specific values. This dependency implies that a simple enumeration of all value combinations is insufficient; instead, systematic dynamic exploration is required.

Example 1.2. Consider now another program where two threads access a shared variable `x`, whose initial value is `0`. Once again, `a` is a thread-local register.

p	q
`a := x;`	`a := x;`
`x := a + 1`	`x := a + 1;`
	`a := x`

We start with an arbitrary interleaving:

$$p_1 q_1 p_2 q_2 q_3$$

Here, the reads p_1 and q_1 read the value `0` and the read q_2 reads the value `1`. At this point, we can infer that these reads might also have read the values `0` and `1`, but we have no evidence that any other values are feasible. However, the following interleaving demonstrates the existence of additional values:

$$p_1 p_2 q_1 q_2 q_3$$

This interleaving, which the algorithm executes after the initial interleaving to cause q_1 to read `1` instead of `0`, introduces the new value `2`, which could not have been discovered from the initial execution. Therefore, the exploration algorithm has to incrementally generate and execute new interleavings that reveal additional feasible values.

Constructing a feasible execution for a given combination of read values poses further challenges. A read can read a particular value from a shared variable only if there exists a corresponding write that:

1. Writes that value to the variable.
2. Occurs before the read in the execution.
3. Is not subsequently overwritten by another write before the read occurs.

However, determining whether such an execution exists has been shown to be NP-complete [11]. Nonetheless, we introduce the ConstructWitness algorithm, which constructs SC executions in polynomial time for nearly all inputs.

Moreover, some combinations of read values may be infeasible, meaning that no valid execution corresponds to them. The above example also demonstrates this. Observe that the read q_3 can never read `0` and the reads p_1 and q_1 can never read `2`. An effective exploration algorithm must be able to detect and discard such infeasible combinations.

1.1 Relation to Previous Work

Before our work, others have employed similar techniques. The general approach of systematically executing a representative subset of all possible interleavings so as to cover all feasible program behaviors is known as *Stateless Model Checking (SMC)* [12]. Existing SMC algorithms [10, 1, 3, 17, 7, 5, 6, 8, 18, 15, 4] typically define an *equivalence relation* between executions; for example, two executions may be considered equivalent if they share the same happens-before relation. Based on this equivalence relation, executions are grouped into equivalence classes. To cover all program behaviours, a property known as *soundness*, the algorithm needs to explore at least one execution from each equivalence class.

When the algorithm explores *exactly one* execution per equivalence class, we say that the algorithm is *optimal* [1].

A key observation underlying these approaches is that whenever two executions are deemed equivalent, all read operations in both executions read the same values. Building on this insight, this thesis explores a more coarse notion of equivalence: we consider two executions equivalent if and only if every read operation reads the same value. We will refer to this notion as *view equivalence*.

The fundamental difference between our approach and prior ones lies in the strength of the equivalence relation. While existing equivalence relations guarantee that equivalent executions read the same values, the converse does not hold: executions that produce identical read values may still be treated as belonging to different equivalence classes. Consequently, earlier algorithms often explore many redundant executions corresponding to the same observable program behavior, often exponentially more. In contrast, our view equivalence-based approach eliminates these redundancies, substantially reducing the exploration space and improving performance.

1.2 Contributions

This thesis presents four distinct contributions that together advance the theory and practice of verifying concurrent programs:

1. **Sound and Optimal Stateless Model Checking Algorithm under View Equivalence for Sequential Consistency.**

   We introduce ViewXplore, the first algorithm for *Stateless Model Checking under View Equivalence for Sequential Consistency*. The algorithm is both *sound*, exploring all feasible program behaviors (at least one execution per view equivalence class), and *optimal*, exploring exactly one execution per view equivalence class.

2. **Efficient Construction of Sequentially Consistent Executions.**

   We introduce ConstructWitness, an algorithm for constructing Sequentially Consistent (SC) executions. Although this problem is known to be NP-complete [11], the algorithm achieves polynomial-time performance for nearly all inputs. It consists of:

   (a) a complete constraint-solving algorithm, and

   (b) two polynomial-time, graph-based heuristics (URF and INCR) that eliminate the need for constraint solving for nearly all inputs.

3. **Implementation.**

   We implement the algorithm and integrate it with the LLVM IR-based Nidhugg stateless model checker. The tool can take C/Pthreads programs as input.

4. **Complexity bound.**

   We establish that the decision version of the model checking problem under *sequential consistency* — *whether any SC execution violates a given safety property* — is NP-complete.

   This result holds independently of the chosen equivalence relation and irrespective of whether *Stateless Model Checking* or any other methodology is employed.

2 Background

2.1 Notation

Basic entities. Threads are denoted by $p, q, r, ...$, memory objects (addresses or shared variables) by $\mathtt{x}, \mathtt{y}, \mathtt{z}, ...$, local variables by $\mathtt{a}, \mathtt{b}, \mathtt{c}, ...$, and values by $\mathtt{0}, \mathtt{1}, ...$ or $\mathtt{v}$.

Operations and executions. An *operation* e is a tuple

$$e = \langle \mathtt{kind},\ \mathtt{thread},\ \mathtt{address},\ \mathtt{value} \rangle$$

where:

- $\mathtt{kind} \in \{\mathsf{R}, \mathsf{W}\}$ indicates whether the operation is a read or a write,
- `thread` is the identifier of the thread that executes the operation,
- `address` denotes the memory object (or variable) being read or written,
- `value` is the value read or written.

Reads are denoted by e_r and writes by e_w.

An *execution* E is a finite sequence of operations.

The set of participating threads is denoted by $\mathsf{Threads}(E)$:

$$\mathsf{Threads}(E) = \{e.\mathtt{thread} \mid e \in E\}$$

Program-order projections and sequence order. For an execution (or, more generally, a sequence of operations) E, the p-projection E_p is the subsequence of E consisting of the operations executed by thread p.

We write $e_1 \xrightarrow{E} e_2$ when e_1 precedes e_2 in the sequence E (i.e., the index of e_1 in E is smaller than that of e_2).

Reads-from and initial writes. Sequential consistency (SC) is assumed throughout. If a read e_r executes immediately after a prefix E' of an execution E, then $e_r.\mathtt{value}$ equals the value of the latest write $e_w \in E'$ such that $e_w.\mathtt{address} = e_r.\mathtt{address}$. For convenience, we assume every read has a source write: if needed, an explicit *initial write* to each address is present at the start of the execution.

Write sets. For a (context-implied) execution E, define

$$\mathcal{W}(\mathtt{addr}, v) \ = \ \{\, e_w \in E \mid e_w \text{ is a write},\ e_w.\mathtt{address} = \mathtt{addr},\ e_w.\mathtt{value} = v \,\},$$

$$\mathcal{W}(\mathtt{addr}) \ = \ \{\, e_w \in E \mid e_w \text{ is a write},\ e_w.\mathtt{address} = \mathtt{addr} \,\}.$$

2.2 The Sequential Consistency Model

The execution of a concurrent program is, by nature, non-deterministic, primarily due to the unpredictable interleaving of threads and the inherent non-determinism in their communication and synchronization.

A widely adopted model for shared-memory concurrency is *Sequential Consistency* (SC) [20], which intuitively enforces single-copy semantics. Under this model, threads issue memory operations. The memory system then arbitrarily selects one thread, processes its memory operation, and allows that thread to issue a subsequent operation. This process continues iteratively until all memory operations have been selected and all threads have terminated. In essence, SC stipulates that the operations of all threads can be linearized into a single global total order such that:

1. The program order of each thread is preserved.
2. Each read reads the value of the most recent write to the same address in the total order.

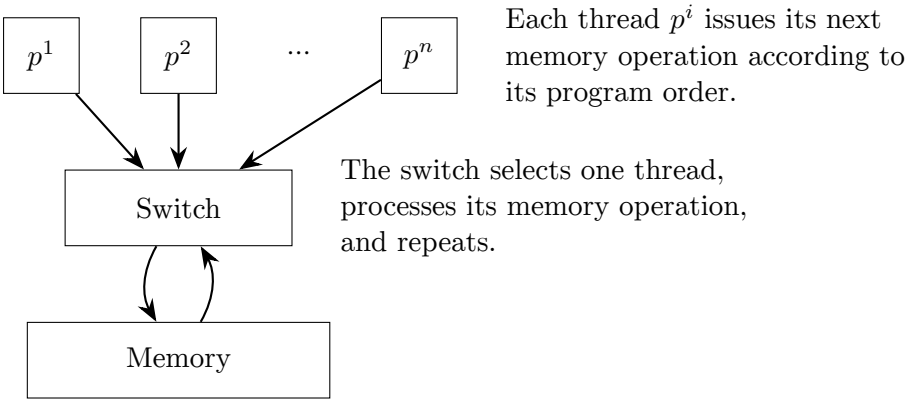

Figure 1: Sequential Consistency Memory Model

Importantly, the interleaving of memory operations from different threads, as determined by the memory system, is non-deterministic. Concurrent programs must be designed to preserve their intended semantics under all interleavings.

2.2.1 Key Relations of Sequential Consistency

We view an execution E as an ordered list of *operations* (reads or writes) issued by a set of threads $\mathsf{Threads}(E)$. Four standard relations – which refine $\xrightarrow{E}$ – capture how these operations depend on one another.

Program order (po). For every thread p, operations appear in the order they are issued.

We define $e_1 \xrightarrow{\texttt{po}} e_2$ if and only if:

1. $e_1 \xrightarrow{E} e_2$
2. $e_1.\texttt{thread} = e_2.\texttt{thread}$

Reads-from (rf). A read e_r reads the value written by the *latest* write e_w to the same location that precedes it in the execution sequence.

We define $e_w \xrightarrow{\texttt{rf}} e_r$ if and only if:

1. $e_w \xrightarrow{E} e_r$
2. e_w is a write operation and e_r is a read operation.
3. e_w and e_r access the same memory object: $e_w.\texttt{address} = e_r.\texttt{address}$.
4. e_w is the latest write to $e_r.\texttt{address}$ before e_r. More formally, there does not exist a write $e'_w \in E$ such that:
   (a) $e'_w.\texttt{address} = e_w.\texttt{address} = e_r.\texttt{address}$
   (b) $e_w \xrightarrow{E} e'_w \wedge e'_w \xrightarrow{E} e_r$

Each read therefore has exactly one incoming rf edge, while a write may have multiple outgoing rf edges to the reads that read from it.

Intuitively, the rf relation links each read to the write from which it obtains its value.

Coherence order (co). For each memory location, all writes to that location form a total order, corresponding to the order in which they reach memory.

We define $e_{w1} \xrightarrow{\texttt{co}} e_{w2}$ if and only if:

1. $e_{w1} \xrightarrow{E} e_{w2}$
2. e_{w1}, e_{w2} are write operations.
3. e_{w1} and e_{w2} access the same memory object: $e_{w1}.\texttt{address} = e_{w2}.\texttt{address}$

Reads-before (rb). If e_r reads-from a write e_w, and a later write e'_w overwrites the same memory object, then e'_w must follow e_r.

We define rb as the composition $\texttt{rf}^{-1}; \texttt{co}$.

In other words, rb links a read to the next write that overwrites the memory object it read.

We also define the happens-before relation hb as the union of the relations po, rf, co, and rb:

$$\texttt{hb} = \texttt{po} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb}$$

2.2.2 An Equivalent Graph-Based Characterisation of Sequential Consistency

Lamport's original definition [20] states that an execution is *sequentially consistent* (SC) if there exists a single global order of events that:

1. preserves every thread's program order, and
2. ensures that each read reads the value written by the latest preceding write to the same address.

Lahav and Vafeiadis [19] showed that Lamport's global-order definition of SC can be reformulated in terms of the existence of suitable rf (reads-from) and co (coherence order) relations such that the induced happens-before dependency graph is acyclic. Their result can be stated as follows.

Theorem 2.1 (SC Acyclicity Theorem [19]). *Let $\widehat{E}$ be a set of operations.*

Let po be an acyclic relation on E such that for all $e_1, e_2 \in E$, if $e_1.\texttt{thread} = e_2.\texttt{thread}$, then either $e_1 \xrightarrow{\texttt{po}} e_2$ or $e_2 \xrightarrow{\texttt{po}} e_1$ (i.e. po totally orders the operations of each thread).

There exists a (Sequentially Consistent) execution E that contains exactly the operations $\widehat{E}$ if and only if there exist:

1. *An acyclic relation* co *such that for all $e_{w1}, e_{w2} \in E$, if e_{w1} and e_{w2} are write operations and $e_{w1}.\texttt{address} = e_{w2}.\texttt{address}$, then either $e_{w1} \xrightarrow{\texttt{co}} e_{w2}$ or $e_{w2} \xrightarrow{\texttt{co}} e_{w1}$.*

   In other words, co *totally orders the write operations for each memory object.*

2. *An acyclic relation* rf *such that for all $e_r \in E$, if e_r is a read operation then there exists a unique operation $e_w \in E$ such that $e_w \xrightarrow{\texttt{rf}} e_r$. This operation e_w must be a write operation that satisfies $e_w.\texttt{address} = e_r.\texttt{address}$ and $e_w.\texttt{value} = e_r.\texttt{value}$.*

   Notice that no additional constraints are imposed on rf*.*

such that the relation

$$\texttt{hb} = \texttt{po} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb} \qquad \textit{with} \quad \texttt{rb} = \texttt{rf}^{-1}; \texttt{co}$$

is acyclic.

Any topological sort of hb *is a (Sequentially Consistent) execution E.*

2.3 Stateless Model Checking

One way to verify that concurrency issues do not arise under a particular interleaving is to systematically explore all possible interleavings, or at least a representative subset of them, as we describe below. This is the goal of *Stateless Model Checking* (SMC) [12].

A central principle of SMC is *stateless execution*. Statelessness means that executions are never paused, stored, or later resumed. Instead, every execution begins from the program's initial state and proceeds without interruption until termination, or until all threads are blocked. This restriction is fundamental to both the design and the correctness of SMC algorithms.

SMC algorithms interact with programs solely through a *Scheduler Oracle*, denoted Scheduler, which controls thread interleaving. The algorithm invokes Scheduler with a sequence of threads, called an *execution prefix*. When invoked, Scheduler starts from the initial state of the program and schedules one operation from a thread at a time, following the order specified in the given prefix. When the prefix is exhausted, if there are still active threads, Scheduler continues the execution by selecting threads in an arbitrary order until all threads have terminated. Upon termination, Scheduler returns the resulting complete *execution* to the model checker.

An *execution* is represented as a sequence of operations on shared memory. For every read or write, the sequence records the operation type, the executing thread, the accessed address, and the value read or written. Thread-local state and operations are omitted, since, under thread determinism, they are determined by shared-memory accesses.

This interaction can be expressed informally as

$$\textsc{Scheduler} : \text{Execution Prefix} \mapsto \text{Execution } (= \text{ Operation}^*)$$

with the understanding that Scheduler is not a mathematical function: the arbitrary scheduling choices made after the prefix is exhausted may lead to multiple possible executions for the same execution prefix.

In the C/Pthreads environment, which we use in our implementation, thread-local memory is not supported, and the entire memory is considered shared. In this setting, ignoring thread-local state has no practical effect. However, in environments that do support thread-local memory, excluding it from executions can yield substantial performance improvements.

Finally, no partial execution state is retained between successive oracle invocations. Allowing paused or resumed executions would require capturing and maintaining full snapshots of the shared memory and the local state of every thread. Since the size of this data is unbounded in general and may grow with the computation itself, storing multiple snapshots of paused executions would quickly lead to prohibitive memory overhead.

In summary, the interaction between the model checker and the program is restricted to a single mechanism: the Scheduler Oracle SCHEDULER. The model checker provides an execution prefix, the oracle executes the program from its initial state until completion, and the resulting execution is returned as a sequence of operations.

Note: In this model, threads are considered deterministic; that is a thread exhibits identical behaviour whenever its reads read the same sequence of values.

The number of possible interleavings under Sequential Consistency (SC) grows exponentially with the number of threads and operations. Even small concurrent programs generate enormous execution spaces, a phenomenon known as the *state explosion problem*. The following theorem formalizes this growth.

Theorem 2.2 (Number of SC interleavings with k threads, n operations each)**.** *Let k threads execute exactly n shared-memory operations each. Then the number of distinct SC executions is*

$$\frac{(kn)!}{(n!)^k} = \binom{kn}{\underbrace{n, n, \dots, n}_{k \text{ times}}}$$

Proof. An SC execution is a total order of all kn operations that respects each thread's program order. That is, for every thread p with operations $(e_{p,1}, e_{p,2}, \dots, e_{p,n})$, the global order must contain

$$e_{p,1} \xrightarrow{E} e_{p,2} \xrightarrow{E} \cdots \xrightarrow{E} e_{p,n}.$$

The constraint is that these operations must appear in order, but they may be interleaved arbitrarily with operations from other threads.

Equivalently, constructing an execution amounts to choosing the positions of each thread's operations in the global sequence of length kn:

- Thread p^1 must occupy n of the kn slots, which can be chosen in $\binom{kn}{n}$ ways.
- After these positions are fixed, thread p^2 occupies n of the remaining $(k-1)n$ slots, giving $\binom{(k-1)n}{n}$ choices.
- This continues until the last thread, which has only n slots remaining and thus only one valid placement.

Because the internal order of each thread is predetermined, there is no additional freedom once the slots of all threads are chosen.

Multiplying the choices for all threads yields:

$$\binom{kn}{n}\binom{(k-1)n}{n}\cdots\binom{n}{n} = \frac{(kn)!}{(n!)^k} = \binom{kn}{\underbrace{n, n, \ldots, n}_{k \text{ times}}}$$

□

Example For $k = 2$ and $n = 10$:

$$\frac{(2 \cdot 10)!}{(10!)^2} = 184{,}756$$

For $k = 3$ and $n = 10$:

$$\frac{(3 \cdot 10)!}{(10!)^3} = 5{,}550{,}996{,}791{,}340$$

i.e., over 5.5 trillion distinct interleavings.

2.4 Dynamic Partial Order Reduction

2.4.1 Happens-Before Equivalence and the Optimal-DPOR Algorithm

A key insight is that many interleavings permitted by the memory model yield the same program behavior, and thus need not all be explored.

For example, suppose thread p writes to a shared variable, and threads q and r subsequently read it. The order in which q and r perform their reads is irrelevant: both read the same value.

Likewise, if two threads perform writes to disjoint variables, and two other threads subsequently read them, the relative ordering of the writes is irrelevant, as the reads read the same values under both orderings.

These examples suggest that a property determining program behavior is the *happens-before* relation between operations.

For convenience, we provide an equivalent definition of the happens-before relation. For an execution E, we define $e_1 \xrightarrow{\text{hb}} e_2$ if and only if:

1. $e_1 \xrightarrow{E} e_2$
2. $e_1.\texttt{address} = e_2.\texttt{address}$
3. At least one of e_1, e_2 is a write operation.

It is evident that if two executions have the same happens-before relation, then all reads read the same values. Thus, assuming each thread behaves deterministically with respect to its inputs (i.e., the values it reads), thread behaviour is fully determined by the happens-before relation.

Therefore, we define two executions E_1 and E_2 to be equivalent if they induce the same happens-before relation. This equivalence relation induces a partitioning of the executions into equivalence classes. All executions belonging to the same equivalence class are equivalent, and it suffices to execute only one of them. Since exploring multiple equivalent executions provides no additional benefit, it is preferable to explore exactly one representative execution per equivalence class to avoid redundant computation.

Two events are defined as *logically concurrent* if they are not related by the transitive closure of the

happens-before relation. The reduction in execution space follows from the fact that the relative order of *logically concurrent* events is irrelevant to program behavior. Thus, executions that differ only in the order of such events are equivalent and need not all be explored.

An algorithm that runs exactly one execution from each equivalence class according to the happens-before equivalence relation is the *Optimal Dynamic Partial Order Reduction* (Optimal-DPOR) algorithm [1, 3].

Classic Dynamic Partial Order Reduction equates executions that share the full happens-before relation $\texttt{hb} = \texttt{po} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb}$. Subsequent work has shown that many of these edges are irrelevant to program behaviour and can be safely omitted without eliminating any feasible program behaviour. The two most prominent refinements are outlined below.

2.4.2 Observer Equivalence and the Optimal-DPOR-Observers Algorithm

Intuitively, a write–write edge matters only if some read actually reads the *later* write. Formally, for an execution E, define the *observer happens-before* relation $e_1 \xrightarrow{\texttt{hb}_{\text{Observers}}} e_2$ if and only if:

$$e_1 \xrightarrow{\texttt{po}} e_2 \text{ or } (e_1 \xrightarrow{\texttt{co}} e_2 \text{ and there exists a read operation } e_r \text{ such that } e_2 \xrightarrow{\texttt{rf}} e_r)$$

More intuitively, $e_{w1} \xrightarrow{\texttt{hb}_{\text{Observers}}} e_{w2}$ holds, beyond program order, only if some read *actually reads from* e_{w2}.

Definition 2.1 (Observer Equivalence)**.** *Executions E_1 and E_2 are* observer equivalent *if and only if:*

$$\texttt{hb}_{\text{Observers}\,E_1} = \texttt{hb}_{\text{Observers}\,E_2}$$

Algorithm. **Optimal-DPOR-Observers** [7] explores exactly one execution per observer-equivalence class. It extends classic DPOR with a modified backtracking rule that ignores write–write races not witnessed by any read, achieving provably sound and optimal exploration with substantially fewer executions.

2.4.3 Reads-From Equivalence and the Optimal SMC-RF Algorithm

Observer equivalence still records the relative order of *observed* writes. An even coarser equivalence relation is to retain *only the reads-from mapping itself*:

Definition 2.2 (Reads-From Equivalence)**.** *Executions E_1 and E_2 are* reads-from equivalent *if and only if*

$$\texttt{rf}_{E_1} = \texttt{rf}_{E_2}$$

Equivalently, every read obtains its value from the same write in both executions.

Since $\texttt{rf}$ fully determines the values returned to every thread, reads-from equivalence implies identical program behaviour.

Algorithm. **Optimal-SMC-RF** [5] achieves sound and optimal exploration under this coarser equivalence relation. Starting from a concrete SC execution, the algorithm mutates the source write

of one read at a time and incrementally repairs the execution using lightweight graph operations. Although deciding whether there exists an SC execution with a candidate $\texttt{rf}$ relation is NP-complete [11], the authors show that simple graph-based heuristics are sufficient to decide for almost all cases in practice, with exhaustive search needed only as a last resort.

2.4.4 Reads-Value-From (RVF) Equivalence and the SMC-RVF Algorithm

Agarwal et al. [6] introduce the *reads-value-from* equivalence, which refines view equivalence by capturing a light-weight causal ordering among reads, while remaining coarser than *reads-from* equivalence.

Formally, let E be an execution, with program-order $\texttt{po}_E$ and reads-from $\texttt{rf}_E$ relations. Define the *causal relation*

$$\texttt{causal}_{\texttt{E}} = (\texttt{po}_{\texttt{E}} \cup \texttt{rf}_{\texttt{E}})^{+}$$

as the transitive closure of their union. The *reads-restricted causal relation* $\texttt{causal}_{\texttt{E}}^{\texttt{Reads}}$ is then obtained by restricting $\texttt{causal}_{\texttt{E}}$ to read events:

$$e_1 \xrightarrow{\texttt{causal}_{\texttt{E}}^{\texttt{Reads}}} e_2 \quad \iff \quad e_1, e_2 \text{ are read operations} \quad \wedge \quad (e_1 \xrightarrow{\texttt{causal}_{\texttt{E}}} e_2).$$

Two complete executions E_1 and E_2 are defined to be *reads-value-from (RVF) equivalent* if:

1. They are view equivalent: every read r reads the same value in both E_1 and E_2.
2. Their *reads-restricted causal relations* are equal: $\texttt{causal}_{\texttt{E}_1}^{\texttt{Reads}} = \texttt{causal}_{\texttt{E}_2}^{\texttt{Reads}}$.

Agarwal et al. [6] present a sound exploration algorithm under this equivalence, but it is *not* optimal – multiple executions from the same reads-value-from equivalence class may be executed.

By contrast[2], ViewXplore:

1. Is based on view equivalence, **coarser** than RVF equivalence, yielding fewer equivalence classes.
2. Guarantees **optimality**: *exactly one* representative per equivalence class is executed.

2.4.5 Data-Centric Dynamic Partial Order Reduction

Similar to Optimal SMC-RF, Data-Centric DPOR [8] is founded on reads-from equivalence. However, Data-Centric DPOR (DC-DPOR) provides no general optimality guarantees. Its optimality holds only in the very restricted case where the underlying communication graph – whose vertices represent program threads and where an edge connects two vertices if they access a common shared variable – is acyclic. In all other situations, DC-DPOR may explore a large number of partial executions that are ultimately determined to be redundant.

2.5 Properties of SMC/DPOR Algorithms

2.5.1 Soundness and Optimality

As we will see later, the previous idea can also be applied to coarsenings of the happens-before relation $\sim$, thereby reducing the number of equivalence classes and consequently the time and space required.

[2]The concepts of View Equivalence and Optimality will be introduced in a subsequent section. However, this point in the thesis provides the most appropriate context for comparing our approach with SMC-RVF.

In what follows, we provide definitions for SMC algorithms parameterized by the choice of equivalence relation $\sim$.

It is essential that an SMC algorithm explores at least one execution from each equivalence class; otherwise, some program behaviours may be lost. This property is called *soundness*.

Definition 2.3 (Soundness of an SMC algorithm)**.** *Let $\mathcal{E}$ be the set of executions run by the* SCHEDULER *once the algorithm has terminated.*

An SMC algorithm is sound *with respect to an equivalence relation $\sim$ if, for every equivalence class induced by $\sim$ on the set of all possible executions (under all thread interleavings), there exists an execution $E \in \mathcal{E}$ that belongs to that class.*

Note that *soundness* requires that *at least one*, but not necessarily exactly one, execution from each equivalence class is explored.

A central design goal for any SMC algorithm is to minimize redundant calls, ensuring that *exactly one* execution is explored from each equivalence class.

Definition 2.4 (Optimality of an SMC algorithm)**.** *Let $\mathcal{E}$ be the set of executions run by the* SCHEDULER *once the algorithm has terminated.*

An SMC algorithm is optimal *with respect to an equivalence relation $\sim$ if, for every equivalence class induced by $\sim$ on the set of all possible executions (under all thread interleavings), there exists a unique execution $E \in \mathcal{E}$ that belongs to that class.*

In an optimal SMC algorithm, the number of SCHEDULER invocations equals the number of equivalence classes under $\sim$.

Optimality is crucial for scalability, as it prevents exponential blow-up from redundant interleavings that yield the same program behaviour.

2.5.2 Performance

The overall performance of a Stateless Model Checking (SMC) algorithm can be approximated by the product:

$$\text{Total Time} = \#(\text{Equivalence Classes}) \times \#(\text{Executions}) \text{ per Equivalence Class} \times \text{Time per Execution}$$

This simple relation highlights the following key trade-offs:

- **Coarser equivalence relations** typically reduce, often exponentially, the number of equivalence classes to be explored.
- **Optimal algorithms** guarantee that exactly one execution is explored per equivalence class, whereas non-optimal algorithms may revisit the same class exponentially many times.
- **Coarser equivalence and optimality** generally incur higher computational overhead, as they require more complex reasoning per execution, thereby increasing the time per execution.

Remark. For some programs, different equivalence relations yield the same partitioning of executions and thus the same number of equivalence classes. In such cases, the overall runtime of an SMC

algorithm may increase if other factors – such as the time required per execution or the number of executions per equivalence class – increase.

2.5.3 NP-completeness of Constructing Sequentially Consistent Executions

A central challenge in reads-from equivalence, and one that becomes even more pronounced in this work, is as follows.

At first sight, one might expect that reconstructing a sequentially consistent execution is straightforward once the *reads-from* mapping is known: take the program-order edges, add an edge $e_w \xrightarrow{\texttt{rf}} e_r$ for every read e_r, perform a topological sort, and the result should be a valid execution. The graph-based characterisation from SC Acyclicity Theorem (Theorem 2.1) shows why this intuition fails.

In particular, an SC execution E can be constructed from a set of operations if there exists proper relations $\texttt{rf}$ and $\texttt{co}$ such that $\texttt{hb} = \texttt{po} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb}$ is acyclic, where $\texttt{rb} = \texttt{rf}^{-1}; \texttt{co}$.

Fixing $\texttt{rf}$ resolves only part of the problem. $\texttt{po}$ is predetermined by the program, so once $\texttt{rf}$ is chosen the remaining freedom lies entirely in the $\texttt{co}$ edges – and, transitively, in the $\texttt{rb}$ edges they induce.

Any topological sort of $\texttt{po} \cup \texttt{rf}$ implicitly commits to one particular coherence order $\texttt{co}$. That choice may introduce a cycle once the corresponding $\texttt{rb} = \texttt{rf}^{-1}; \texttt{co}$ edges are added.

The acyclicity criterion is thus far stronger: *does there exist* ***some*** *coherence order* $\texttt{co}$ *such that the combined graph* $\texttt{hb} = \texttt{po} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb}$ *is acyclic?*

Informally, a read e_r that reads-from a write e_w excludes every other write to the same location from appearing between e_w and e_r in the global order. However, a topological sort of $\texttt{rf} \cup \texttt{co}$ alone does not enforce this constraint.

Gibbons and Korach [11] proved that the following decision problem is NP-complete:

> *Given a set of operations $\widehat{E}$ – where the values read by read operations are fixed – does there exist a Sequentially Consistent Execution E that contains exactly the operations $\widehat{E}$?*

The result also holds when a desired $\texttt{rf}$ relation is provided as input, as in the SMC-RF algorithm. In this setting, the only remaining nondeterminism is to select a coherence order $\texttt{co}$ that keeps $G = \texttt{po} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{rb}$ acyclic. The need to search over all such $\texttt{co}$ choices drives the NP-completeness.

Under *view equivalence*[3], reconstruction is strictly harder, and is exactly what the above decision problem captures. Here the reads-from relation is not fixed a priori. We must jointly explore the space of $(\texttt{rf}, \texttt{co})$ pairs to find one that avoids a cycle in $\texttt{hb}$. This expanded search space strictly contains the subproblem of construction under reads-from equivalence, underscoring the algorithmic difficulty of constructing witnesses at the granularity required by view equivalence.

Practical workaround. Optimal SMC for reads-from equivalence mitigates this complexity with heuristics that incrementally repair an existing execution. Our approach adopts a similar philosophy, but the challenge is more severe under *view equivalence*, since the reads-from relation is not predetermined.

[3]The formal definition of view equivalence will appear shortly; for now, the important point is that the reads-from relation is not fixed.

3 View Equivalence

3.1 Definition

We introduce a new equivalence relation on executions, termed *View Equivalence*. This equivalence relation is coarser than previously studied relations, including those based on happens-before [10, 1, 3], observer equivalence [7], and reads-from [5]. Its coarseness enables significant reduction in the number of executions explored, by collapsing multiple indistinguishable executions into a single equivalence class, while still preserving all program behaviours.

Definition 3.1 (View Equivalence)**.** *Two executions are* ***view equivalent*** *if they contain the same set of read operations, and each read reads the same value in both executions.*

In essence, View Equivalence leverages thread determinism (a thread exhibits the same behaviour whenever it reads the same values) to its fullest extent.

Example 3.1. Consider three threads sharing a variable `x`, initially set to `0`:

$$\begin{array}{ccc} p & q & r \\ \texttt{x := 0} & \Big\| \; \texttt{x := 0} & \Big\| \; \texttt{a := x} \end{array}$$

The following six interleavings are possible:

1. $r_1 p_1 q_1$
2. $r_1 q_1 p_1$
3. $p_1 r_1 q_1$
4. $q_1 p_1 r_1$
5. $q_1 r_1 p_1$
6. $p_1 q_1 r_1$

In all six interleavings, the single read operation r_1 returns the same value `0`. Therefore, all six executions belong to the same View Equivalence class.

The key intuition is that since all reads observe the same values, the behaviour of every thread is identical across all six interleavings. Hence, it suffices to explore just one of them. This is the essence of View Equivalence.

By contrast, under reads-from equivalence the situation differs. In interleavings (1) and (2), r_1 reads from the initial write. In (3) and (4), r_1 reads from p_1, and in (5) and (6), it reads from q_1. Thus, there are three distinct reads-from equivalence classes.

Consequently, an optimal Stateless Model Checking algorithm under View Equivalence executes only one execution, whereas an optimal algorithm under Reads-From Equivalence must explore three executions, incurring additional cost.

Finally, note that no stateless model checking algorithm that ignores values can safely reduce the number of executions below three in this example. The reason is that p_1 and q_1 could have written different values, in which case r_1 would read different values in the three reads-from classes, potentially altering the behaviour of thread r.

3.2 Exponential Reduction in View Equivalence Classes

We present a set of programs demonstrating that *view equivalence* can induce significantly fewer equivalence classes than previously studied notions such as happens-before equivalence and reads-from equivalence.

3.2.1 The ManySameValue (msv) Program

Program Definition

The program consists of two threads and a single shared variable `x`, initially set to `0`, and is parameterized by a positive integer n. Thread p performs n writes of the value `0` to `x`, while thread q performs n reads from `r`:

No.	p	q
1	`x := 0;`	$\mathtt{a}_1$ `:= x;`
2	`x := 0;`	$\mathtt{a}_2$ `:= x;`
	⋮	
n	`x := 0`	$\mathtt{a}_n$ `:= x`

Despite its simplicity, this program captures behaviors that naturally arise in low-level concurrent code. In particular, repeatedly writing or reading fixed values is not only a standalone pattern but also a building block within more complex synchronization mechanisms, such as locks.

Counting Equivalence Classes

It is easy to see that this program has exactly **one view equivalence class**. Since the initial value of `x` is `0`, and all subsequent writes by p also write the value `0`, every read of q must read the value `0`, regardless of the interleaving. Thus, all executions contain the same set of read operations with identical read values, and are therefore view equivalent.

In contrast, under both the happens-before and reads-from equivalence relations, no two interleavings collapse into the same class. Intuitively, even though every read always reads the same *value*, the specific write from which it reads varies across interleavings. Hence every distinct interleaving produces a different equivalence class. The following theorem formalizes and proves this observation.

Theorem 3.1 (Equivalence classes in the msv program)**.** *In the* msv *program, every execution forms a distinct equivalence class under both reads-from and happens-before equivalence.*

Proof. An execution E is a total order over $\{p_1, \dots, p_n, q_1, \dots, q_n\}$ that preserves the program order of threads p and q.

For an execution E, let $\text{index}_E(e)$ denote the index of operation e in E (indexes start at 1).

Define $k_E(i)$ as the number of writes that appear before the read q_i in E. In particular, $p_{k_E(i)}$ is the latest write in E before the read q_i.

By Sequential Consistency, each read returns the value of the latest write that precedes it in the global order. Hence, the reads-from relation of E is exactly

$$\mathtt{rf}_E = \{(p_{k_E(i)}, q_i) \mid 1 \le i \le n\}$$

Thus $\texttt{rf}_E$ is completely determined by the vector $k_E = (k_E(1), \ldots, k_E(n))$.

We now show that E can be uniquely reconstructed from k_E.

Set $a_0 := k_E(i)$ and, for $1 \le i \le n-1$, set $a_i := k_E(i+1) - k_E(i)$, and finally $a_n := n - k_E(i)$. Each a_i is a nonnegative integer because the sequence $k_E(i)$ is nondecreasing and bounded by n.

Equivalently:

- a_0 is the number of writes before q_1.
- For $1 \le i \le n-1$, a_i is the number of writes strictly between q_i and q_{i+1}.
- a_n is the number of writes after q_n.

Because the per-thread orders $p_1 \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} p_n$ and $q_1 \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} q_n$ are fixed, the counts $(a_0, \ldots, a_n)$ determine the execution E:

$$\underbrace{p_1, \cdots, p_{a_0}}_{a_0 \text{ writes}},\ q_1,\ \underbrace{p_{a_0+1}, \cdots, p_{a_0+a_1}}_{a_1 \text{ writes}},\ q_2,\ \cdots,\ q_n,\ \underbrace{p_{n-a_n+1}, \cdots, p_n}_{a_n \text{ writes}}$$

Hence E is uniquely determined by k_E.

Therefore, if $E \neq E'$, then $k_E \neq k_{E'}$, which implies $\texttt{rf}_E \neq \texttt{rf}_{E'}$, and consequently $\texttt{hb}_E \neq \texttt{hb}_{E'}$.

Thus, each execution corresponds to its own equivalence class under both reads-from and happens-before equivalence relations. □

Corollary 3.1 (Number of equivalence classes in the MSV program)**.** *In the* MSV *program, the number of happens-before equivalence classes and reads-from equivalence classes is:*

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2}$$

Proof. By Theorem 2.2, the number of SC executions with n operations per thread is $\binom{2n}{n} = \frac{(2n)!}{(n!)^2}$.

By Theorem 3.1, each execution is its own equivalence class.

Thus the number of equivalence classes is also $\binom{2n}{n} = \frac{(2n)!}{(n!)^2}$. □

Asymptotic Analysis

Theorem 3.2 (Asymptotic growth of equivalence classes in the MSV program)**.** *It holds that*

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} = \Theta\left(\frac{4^n}{\sqrt{\pi n}}\right) = \omega\Big((4-\varepsilon)^n\Big) \quad \textit{for any } \varepsilon > 0$$

Proof. Applying Stirling's approximation:

$$\log n! = n \log n - n + \frac{1}{2}\log(2\pi n) + O\left(\frac{1}{n}\right)$$

we obtain:

$$\begin{aligned}
&\log\binom{2n}{n} = \log\frac{(2n)!}{(n!)^2} \\
&= \log(2n)! - 2\log n! \\
&= 2n\log 2n - 2n + \frac{1}{2}\log(4\pi n) + O\Big(\frac{1}{n}\Big) - 2\left(n\log n - n + \frac{1}{2}\log(2\pi n) + O\Big(\frac{1}{n}\Big)\right) \\
&\overset{O(\frac{1}{n})-O(\frac{1}{n})=\pm O(\frac{1}{n})}{=} 2n\log\frac{2n}{n} + \frac{\log 2^2 + \log \pi n}{2} - (\log 2 + \log \pi n) \pm O\Big(\frac{1}{n}\Big) \\
&= 2n\log 2 - \frac{1}{2}\log \pi n \pm O\Big(\frac{1}{n}\Big)
\end{aligned}$$

Exponentiating both sides gives:

$$\binom{2n}{n} = 2^{\log\binom{2n}{n}} = 2^{2n}\cdot(\pi n)^{-\frac{1}{2}}\cdot 2^{\pm O(\frac{1}{n})} = \Theta\Big(\frac{4^n}{\sqrt{\pi n}}\Big)$$

Above is used the fact that $2^{O(\frac{1}{n})} = \Theta(1)$.

Finally, for any $\varepsilon > 0$:

$$\Theta\Big(\frac{4^n}{\sqrt{\pi n}}\Big) = \Theta\left((4-\varepsilon)^n \cdot \frac{\left(\frac{4}{4-\varepsilon}\right)^n}{\sqrt{\pi n}}\right) \overset{\frac{4}{4-\varepsilon}>1}{=} \omega\Big((4-\varepsilon)^n\Big)$$

□

Hence, the number of equivalence classes under happens-before and reads-from equivalence grows exponentially with n, whereas under view equivalence, there is only a single equivalence class.

Thus, any sound and optimal exploration algorithm based on happens-before or reads-from equivalence must explore $\Theta\big(\frac{4^n}{\sqrt{\pi n}}\big) = \omega((4-\varepsilon)^n)$ executions, whereas any sound and optimal exploration algorithm based on view equivalence must explore exactly only one.

3.2.2 The ReadInc Program

The ReadInc program consists of n threads and a single shared variable `x`, initialized to `0`. Each thread reads the value of `x` into a local variable `a`, and subsequently writes the value `a + 1` back to `x`, as shown below:

p^1	p^2	$\cdots$	p^n
`a := x;`	`a := x;`		`a := x;`
`x := a + 1`	`x := a + 1`		`x := a + 1`

This program serves a dual purpose. First, it highlights the practical impact of view equivalence on reducing the number of explored executions. Second, it provides a robust test for verifying that a Stateless Model Checking (SMC) algorithm correctly constructs consistent executions: A key insight in this program is that it encodes causal structure implicitly: each thread is only able to write the value $v+1$ after reading the value v. Hence, the value it writes reflects its position in a causal chain of updates to the shared variable `x`. This dependency is enforced entirely through data values, without

the need for explicit control flow or synchronization, making the program a useful tool for debugging and validating SMC implementations.

By Theorem 2.2, the total number of interleavings is given by:

$$\frac{(2n)!}{2^k}$$

At present, we have not derived closed-form expressions or asymptotic bounds for the number of equivalence classes under the happens-before, observers, reads-from or view equivalence relations in this program. However, we report concrete numbers for $n = 2, 3, 4, 5, 6, 7$.

To measure the number of happens-before, observers and reads-from equivalence classes, we use the Nidhugg model checker. For view equivalence, we validate results using two independent methods:

1. Extract the executions produced by Nidhugg's implementation of Optimal under Reads-From Equivalence SMC algorithm (Optimal-SMC-RF) and post-process them by regrouping according to the view equivalence relation.
2. Perform an exhaustive enumeration of all valid interleavings (brute-force), then group them according to view equivalence.

Table 1 and Fig. 2 summarize the number of equivalence classes under each equivalence relation for $n = [2..7]$.

n	Interleavings	Happens-Before	Observers	Reads-From	View
2	6	4	3	3	3
3	90	36	22	16	13
4	2,520	576	281	125	75
5	113,400	14,400	5,566	1,296	541
6	7,484,400	518,400	157,717	16,807	4,683
7	681,080,400	25,401,600	6,053,748	262,144	47,294

Table 1: Number of equivalence classes under various equivalence relations for the ReadInc program.

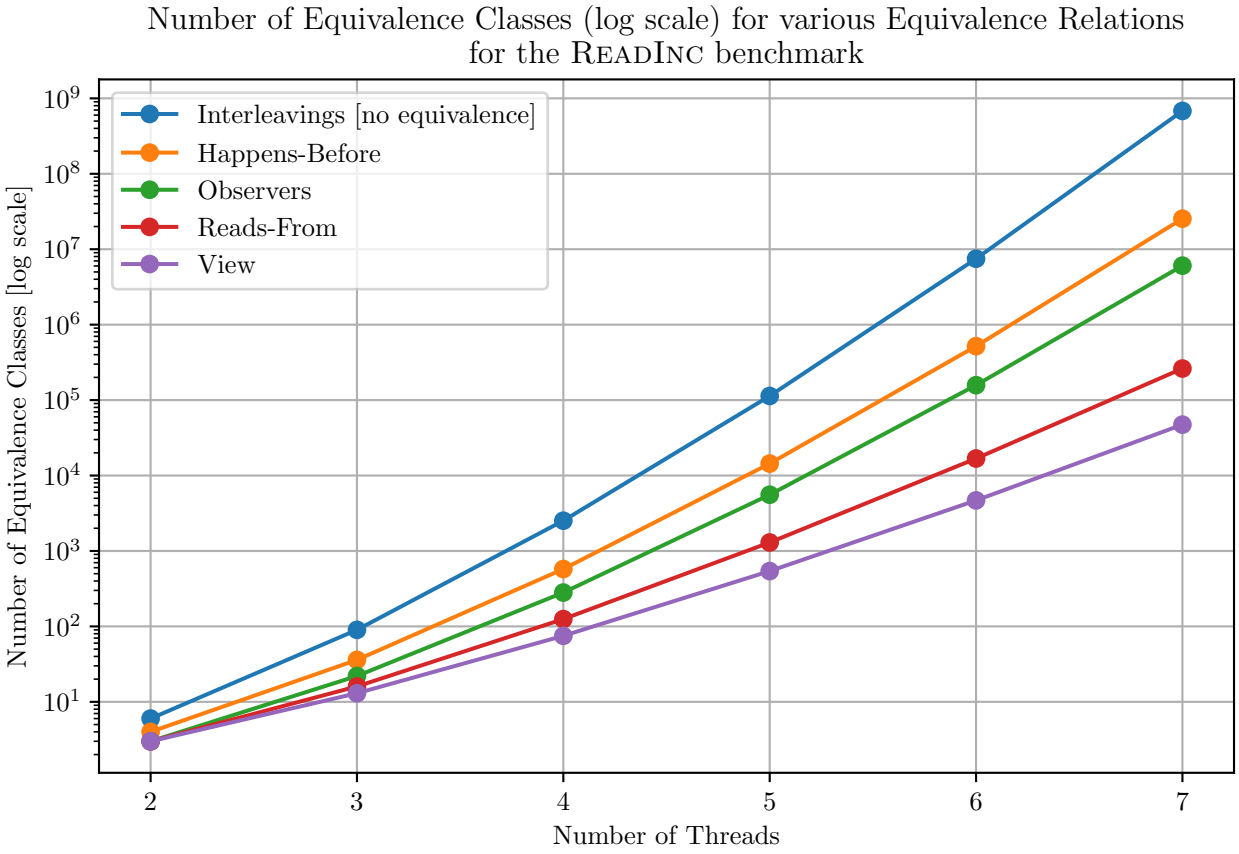

Figure 2: Equivalence class counts (log scale) for the ReadInc program.

Although the logarithmic plot does not make precise growth rates immediately apparent, the noticeably lower slope for view equivalence suggests a significantly slower exponential growth compared to the happens-before, observers, and reads-from equivalence relations.

For completeness, Fig. 3 illustrates the equivalence-class structure of the ReadInc program for $n = 3$. The figure presents one representative execution from each of the 36 happens-before equivalence classes, which are then hierarchically grouped: first into observers equivalence classes (green), next into reads-from equivalence classes (red), and finally into the coarsest view equivalence classes (blue). Each view equivalence class is additionally annotated with a triple $\langle a, b, c \rangle$, indicating the values read by the three threads (p, q, and r), respectively.

View Class 1 ⟨0,1,2⟩

Reads-From Class 1

Observers Class 1

$p_1p_2q_1q_2r_1r_2$

View Class 2 ⟨0,2,1⟩

Reads-From Class 2

Observers Class 2

$p_1p_2r_1r_2q_1q_2$

View Class 3 ⟨1,0,2⟩

Reads-From Class 3

Observers Class 3

$q_1q_2p_1p_2r_1r_2$

View Class 4 ⟨2,0,1⟩

Reads-From Class 4

Observers Class 4

$q_1q_2r_1r_2p_1p_2$

View Class 5 ⟨2,1,0⟩

Reads-From Class 5

Observers Class 5

$r_1r_2q_1q_2p_1p_2$

View Class 6 ⟨1,2,0⟩

Reads-From Class 6

Observers Class 6

$r_1r_2p_1p_2q_1q_2$

View Class 7 ⟨0,1,1⟩

Reads-From Class 7

Observers Class 7

$p_1p_2q_1r_1q_2r_2$

$p_1p_2q_1r_1r_2q_2$

View Class 8 ⟨1,0,1⟩

Reads-From Class 8

Observers Class 8

$q_1q_2p_1r_1p_2r_2$

$q_1q_2p_1r_1r_2p_2$

View Class 9 ⟨1,1,0⟩

Reads-From Class 9

Observers Class 9

$r_1r_2p_1q_1p_2q_2$

$r_1r_2p_1q_1q_2p_2$

View Class 10 ⟨0,0,0⟩

Reads-From Class 10

Observers Class 10

$p_1q_1r_1p_2q_2r_2$

$p_1q_1r_1p_2r_2q_2$

$p_1q_1r_1q_2p_2r_2$

$p_1q_1r_1q_2r_2p_2$

$p_1q_1r_1r_2p_2q_2$

$p_1q_1r_1r_2q_2p_2$

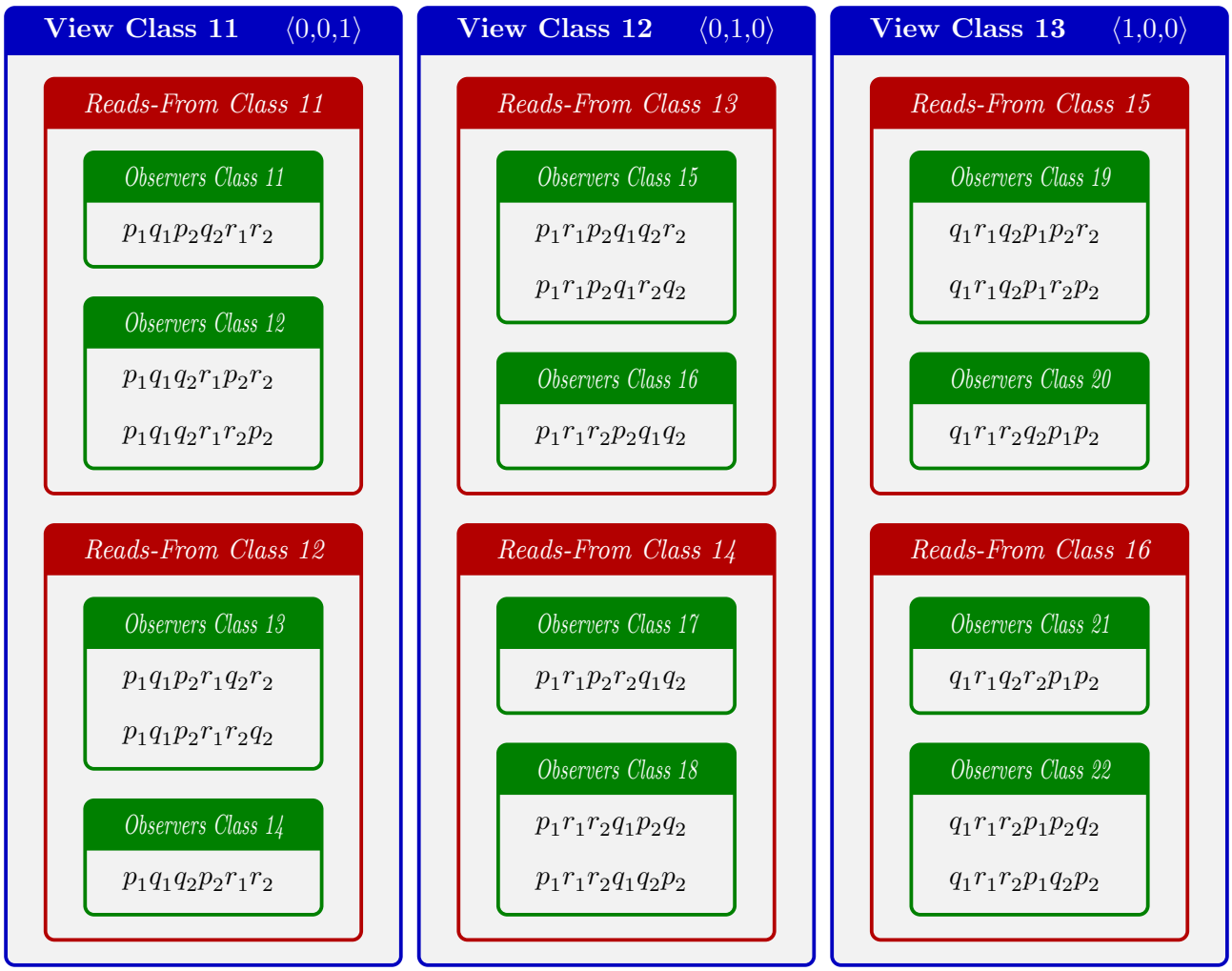

Figure 3: Representative executions from each of the 36 happens-before equivalence classes in READ-INC, hierarchically grouped into observers (green), reads-from (red), and view (blue) equivalence classes. Each view equivalence class is annotated with a triple $\langle a, b, c \rangle$, where a, b, and c denote the values read by threads p, q, and r, respectively.

4 Exploration Algorithm

Our goal is to explore a single execution from each view equivalence class of sequentially consistent executions. The exploration is structured around a combinatorial abstraction, which we term *read-cuts*.

4.1 Definitions

Definition 4.1 (Read-Cut)**.** *Let $\mathcal{T}$ be a set of thread identifiers and* Values *a set of values. A* read-cut *is a function $rc : \mathcal{T} \mapsto \textsc{Values}^*$ that assigns to each thread a sequence of read values.*

Example 4.1. Consider the program with two threads, p and q, and a shared variable `x`, initially set to 0.

p	q
`a := x;`	`a := y;`
`x := 1`	`b := x`

- $rc_1(p) = [0], rc_1(q) = [0]$ means that both p_1 and q_1 read the initial value 0.
- $rc_2(p) = [1], rc_2(q) = [0]$ means p_1 reads 1 and q_1 reads 0.
- $rc_3(p) = [], rc_3(q) = [0]$ means q_1 reads 0 and p_1 is not included in the read-cut.

Definition 4.2 (p-projection of execution E)**.** *Let E be an execution. The p-projection of E, denoted E_p, is the subsequence of E containing all operations of thread p.*

Definition 4.3 (Read-Cut of Execution)**.** *Let E be an execution and rc a read-cut. We say that rc is a* read-cut of E *if for every thread $p \in dom(rc)$, the sequence of values read by the first $|rc(p)|$ read operations of E_p equals $rc(p)$.*

Definition 4.4 (Execution-Induced Read-Cut)**.** *Let E be an execution (or prefix, or any sequence of operations). The* read-cut induced by *E, denoted rc_E, is defined as follows: for each thread $p \in \mathsf{Threads}(E)$, let $rc_E(p)$ be the sequence of values read by all the read operations of p in E, listed in program order. Formally, for every $p \in \mathsf{Threads}(E)$:*

$$rc_E(p) = [\, e.\texttt{value} \mid e.\texttt{thread} = p \wedge e \text{ is a read operation} \,].$$

Notice that two executions E_1, E_2 are *view equivalent* if and only if they induce the same read-cut, i.e., $rc_{E_1} = rc_{E_2}$.

Definition 4.5 (Feasible Read-Cut)**.** *A read-cut rc is* feasible *if there exists an SC execution E with a prefix E' such that $rc = rc_{E'}$.*

Definition 4.6 (Witness)**.** *Let rc be a feasible read-cut. An execution prefix E' is a* witness *for rc if there exists a complete execution E such that E' is a prefix of E and $rc = rc_{E'}$.*

Definition 4.7 (rc-restriction of execution E)**.** *Let E be an execution and rc a read-cut of E. The rc-*restriction of *E, denoted E_{rc}, is the subsequence of E containing all operations $e \in E$ such that e is not after the $(1 + |rc(e.\texttt{thread})|)^{th}$ read operation of $E_{e.\texttt{thread}}$.*

Intuitively, E_{rc} is the maximal subsequence of E that contains exactly the reads specified by rc.

Equivalently, for each thread p, E_{rc} contains all operations of E_p up to (but not including) the first read of E_p that is not *in $rc(p)$.*

It follows immediately that $rc_{E_{rc}} = rc$.

Example 4.2. Consider the program with two threads, p and q, and a shared variable `x`, initially set to 0.

p	q
`x := 1;`	`a := x;`
`a := x;`	`x := 3`
`x := 2;`	
`b := x`	

Let an execution be $E = [q_1 p_1 p_2 p_3 p_4 q_2]$ and consider the read-cut $rc_1(p) = [1], rc_2(p) = [0]$.

The corresponding rc-restriction of E is $E_{rc} = [q_1 p_1 p_2 p_3 q_2]$.

Remark. Let E_1, E_2 be executions with read-cuts rc_1, rc_2, respectively. Because threads are deterministic given fixed read values, if $rc_1 = rc_2 = rc$, then for every $p \in dom(rc)$:

1. the maximal prefix of E_{1_p} containing exactly $|rc(p)|$ reads, and
2. the maximal prefix of E_{2_p} containing exactly $|rc(p)|$ reads

are identical. Therefore, the rc-restrictions $E_{1_{rc}}$ and $E_{2_{rc}}$ contain the same set of operations; their only possible difference lies in the relative ordering of operations from different threads.

4.2 Algorithm

The algorithm explores the space of read-cuts by analyzing executions and generating new executions with unexplored read-cuts.

Its pseudocode is given in Algorithm 1.

A set Explored stores all read-cuts that have already been analyzed (Line 1). A second set ToBeExplored stores read-cuts that have been witnessed by some execution that has run but has not yet been processed (Line 2). Finally, a container Executions stores executions awaiting processing (Line 3). The order in which executions are extracted from this container is left unspecified by the algorithm. In particular, any strategy may be used. The options considered are a stack (yielding DFS) or a queue (yielding BFS). The traversal strategy may affect memory consumption but does not affect correctness. BFS is chosen in order to reduce memory usage.

Initially, the algorithm invokes the scheduler oracle Scheduler with the empty execution prefix (Line 4) and obtains an arbitrary execution E. All read-cuts of E are enumerated and added to ToBeExplored (Line 6), and E is added to Executions (Line 5).

The algorithm then iteratively processes executions from Executions until it becomes empty (Line 7). In each iteration, an execution E is extracted from Executions (Line 8), and all its read-cuts rc are enumerated and considered one by one (Line 9).

If $rc \notin$ ToBeExplored, the algorithm skips it (Line 10), since some other execution E' must already be a witness for rc and rc was analyzed while processing E'. Otherwise, rc is removed from ToBeExplored and added to Explored (Lines 11 and 12). The algorithm then considers all one-read extensions of rc by iterating over threads of E (Line 13).

For each thread $p \in \mathsf{Threads}(E)$:

1. Identify the first read e_r in E_p not included in rc (i.e., the $(1 + |rc(p)|)^{th}$ read of E_p). (Line 14) If no such read exists (all reads of p are already covered), continue. (Line 15)

   Compute the set Values of all values v written to $e_r.\texttt{address}$ by some write e_w, under the constraint that e_w is not after the $(1 + |rc(e_w.\texttt{thread})|)^{th}$ read of $E_{e_w.\texttt{thread}}$ (otherwise e_w would be excluded by the current read-cut restriction of the execution). (Line 16)

   For each $v \in$ Values, define (Line 18) the new read-cut

   $$rc' = rc[p \mapsto rc(p) \cdot v].$$

If rc' has already been encountered (i.e. $rc' \in$ Explored $\vee$ $rc' \in$ ToBeExplored), skip it (Line 19). Otherwise, attempt to construct a witness for rc' using the function ConstructWitness, formally defined below.

Compute the rc-restriction E_{rc} of E (Line 21). Extend E_{rc} with the new read $e'_r = e_r[\texttt{value} \mapsto v]$ (Line 20), producing the sequence E' (Line 22). Call ConstructWitness(E', e'_r) to attempt to construct a valid execution prefix $\widetilde{E'}$ (Line 23). If no such prefix exists (i.e. ConstructWitness(E', e'_r) returned None and thus the read-cut rc' is infeasible), continue. (Line 24)

Otherwise, let $\widetilde{E'}$ be the execution prefix returned by ConstructWitness. Invoke the scheduler oracle Scheduler on input $\widetilde{E'}$ (Line 25). It returns a complete execution $\widehat{E'}$ such that $\widetilde{E'}$ is a prefix of $\widehat{E'}$.

Add $\widehat{E'}$ to Executions (Line 26), and enumerate all its read-cuts $\widehat{rc}$ (Line 27). Add any new $\widehat{rc} \notin$ Explored to ToBeExplored (Line 28).

This process continues until Executions becomes empty (Line 7).

Definition 4.8 (ConstructWitness)**.** *Let E be a sequence of operations representing a* subsequence *of an execution, extended by a newly added read operation e_r.*

The function ConstructWitness(E, e_r) *attempts to construct a reordering $\widetilde{E}$ of E such that $\widetilde{E}$ is a prefix of some sequentially consistent (SC) execution. Equivalently, it attempts to construct a witness for the read-cut rc_E corresponding to this extended sequence, if the read-cut is feasible.*

If such a reordering exists, ConstructWitness(E, e_r) *returns the corresponding SC execution prefix $\widetilde{E}$. Otherwise (i.e., if rc_E is infeasible), it returns* Inconsistent.

Algorithm 1: ViewXplore Exploration Algorithm

```
1  Explored ← ∅                                              // Set of analyzed read-cuts
2  ToBeExplored ← ∅                                  // Read-cuts witnessed but not yet analyzed
3  Executions ← []                                          // Executions awaiting processing
4  E ← Scheduler([])
5  Executions ← [E]
6  foreach read-cut rc of E do ToBeExplored ← ToBeExplored ∪ {rc}
7  while Executions ≠ [] do
8      [E, Executions] ← Executions        // Extraction order may follow any traversal
         (e.g. DFS with a stack, BFS with a queue)
9      foreach read-cut rc of E do
10         if rc ∉ ToBeExplored then continue
11         ToBeExplored ← ToBeExplored \ {rc}
12         Explored ← Explored ∪ {rc}
13         foreach p ∈ Threads(E) do
14             e_r ← (1 + |rc(p)|)^th read in E_p                  // First read of p "not in" rc
15             if e_r = None then continue                  // No remaining reads for thread p
16             Values ← { e_w.value | e_w ∈ E, e_w.address =
                 e_r.address, e_w not after (1 + |rc(e_w.thread)|)^th read of E_{e_w.thread} }
17             foreach v ∈ Values do
18                 rc' ← rc[p ↦ rc(p) · v]
19                 if rc' ∈ Explored or rc' ∈ ToBeExplored then continue
20                 e'_r ← e_r[value ↦ v]
21                 E_rc ← [ e | e ∈ E ∧ e is not after the (1 + |rc(e.thread)|)^th read of E_{e.thread} ]
                     // rc-restriction of E
22                 E' ← E_rc + [e'_r]
23                 Ẽ' ← ConstructWitness(E', e'_r)
24                 if Ẽ' = None then continue
25                 Ê' ← Scheduler(Ẽ')
26                 Executions ← Executions + [Ê']
27                 foreach read-cut r̂c of Ê' do
28                     if r̂c ∈ Explored then continue
29                     ToBeExplored ← ToBeExplored ∪ {r̂c}
```

Enumerating the read-cuts of an execution E can be done as follows. First, compute the read-cut rc_E induced by E. The set of all read-cuts of E is then given by the cross-product over all threads:

$$\prod_{p \in \mathsf{Threads}(E)} \mathrm{Pref}(rc_E(p)),$$

where $\mathrm{Pref}(S)$ denotes the set of all prefixes of a sequence S, including the empty prefix and the full sequence. We will later refine this construction to exclude some infeasible read-cuts.

4.3 Example: Running the Algorithm on ReadInc [$n = 2$]

We illustrate the execution of ViewXplore on the ReadInc program with two threads ($n = 2$). For conciseness, we refer to the two threads as p (instead of p^1) and q (instead of p^2).

We begin by invoking the Scheduler on the empty execution prefix [], and obtain:

$$E_1 = p_1 q_1 p_2 q_2$$

Execution E_1 is added to the container Executions.

Processing E_1. The algorithm then enters the main loop and begins processing E_1. This execution has following read-cuts:

$rc_{1,1}$	$p : [],$	$q : []$
$rc_{1,2}$	$p : [0],$	$q : []$
$rc_{1,3}$	$p : [],$	$q : [0]$
$rc_{1,4}$	$p : [0],$	$q : [0]$

These read-cuts are added to the set ToBeExplored.

Processing the first read-cut $rc_{1,1}$. We begin with the first read-cut and move it from ToBeExplored to Explored. We attempt to expand it by introducing a new read value for each thread:

1. The first read of p not included in $rc_{1,1}$ is p_1. The set of values written to x in E_1 is $\{0, 1\}$. Since p_1 read 0 in E_1, the only alternative value is 1. We thus consider the expanded read-cut $p : [1], \quad q : []$.

   As this read-cut does not belong to either Explored or ToBeExplored, we proceed to examine it. Conceptually, this corresponds to constructing an execution prefix where p_1 reads 1 and q_1 does not exist. This is clearly infeasible.

   The algorithm constructs the $rc_{1,1}$-restriction of E_1, which is the empty sequence, appends the new read operation p_1 reading 1, and invokes ConstructWitness on this sequence. ConstructWitness reports infeasibility, so this read-cut is discarded.

2. Thread q is handled analogously, yielding no feasible expansion.

Processing the second read-cut $rc_{1,2}$. Next, we process the second read-cut by moving it from ToBeExplored to Explored and again attempt to expand it.

1. Thread p has no additional reads available.

2. The first read of q not included in $rc_{1,2}$ is q_1. The values written to x in E_1 are again $\{0, 1\}$, and the alternative to the value read in E_1 (0) is 1. We therefore examine the expanded read-cut $p : [0], \quad q : [1]$.

   Since it is not yet in Explored or ToBeExplored, we proceed. The $rc_{1,2}$-restriction of E_1 is:

$$E_{1,rc_{1,2}} = p_1 p_2$$

We append the new read operation q_1 reading 1, and invoke ConstructWitness on this sequence. This call returns the execution prefix $p_1 p_2 r_2$.

Invoking Scheduler on this prefix yields:

$$E_2 = p_1 p_2 r_1 r_2$$

Execution E_2 has the following read-cuts:

$rc_{2,1}$ $p : [],$ $q : []$
$rc_{2,2}$ $p : [0],$ $q : []$
$rc_{2,3}$ $p : [],$ $q : [1]$
$rc_{2,4}$ $p : [0],$ $q : [1]$

Among these, $rc_{2,1}$ and $rc_{2,2}$ already appear in Explored or ToBeExplored and are thus skipped, while $rc_{2,3}$ and $rc_{2,4}$ are added to ToBeExplored. Finally, E_2 is added to the Executions container.

The remaining read-cuts of E_1 are processed in the same manner.

Subsequent processing. After all read-cuts of E_1 have been explored, the algorithm selects another execution from the Executions container and repeats the process. This continues until Executions becomes empty, at which point the algorithm terminates.

4.4 Optimality

We now establish the optimality of the exploration algorithm under view equivalence.

Invariant 4.1. At any point during the algorithm, all read-cuts of all executions that have run belong to Explored $\cup$ ToBeExplored.

Proof. Immediately after an execution is run, all its read-cuts are added to ToBeExplored. Whenever a read-cut is removed from ToBeExplored, it is added to Explored. Hence the claim always holds. □

The following theorem, together with the soundness proof given later, establishes that the SMC algorithm is optimal under view equivalence.

Theorem 4.1 (Optimality of ViewXplore)**.** *After Algorithm 1 terminates, no two executions E_1 and E_2 that are view equivalent have both run.*

Proof. Assume for contradiction that the algorithm terminates and two executions E_1 and E_2 have run that are view equivalent. Without loss of generality, suppose E_1 was run before E_2.

Since E_1 and E_2 are view equivalent, they induce exactly the same read-cuts. Consider the state of the algorithm immediately after E_1 has run but before E_2 has. By Invariant 4.1, all read-cuts of E_1 must already belong to ToBeExplored $\cup$ Explored. Since E_2 has the same read-cuts as E_1, the same must hold for E_2.

Now consider the call to the scheduler oracle SCHEDULER that produced E_2. This call must have been triggered during the processing of some new read-cut rc', which is a read-cut of E_2. as guaranteed by the definition of CONSTRUCTWITNESS.

By construction, rc' was only explored because it was not already in TOBEEXPLORED $\cup$ EXPLORED at that point. But from the argument above, rc' was in fact already present in TOBEEXPLORED $\cup$ EXPLORED. This contradiction establishes the theorem. □

4.5 Soundness

We now establish the soundness of the exploration algorithm.

Theorem 4.2 (Soundness of VIEWXPLORE)**.** *At termination, the exploration algorithm has visited* at least one *execution from each* view equivalence *class induced by some feasible sequentially consistent execution.*

Proof. Throughout the exploration, we maintain a partition of the universe of view equivalence classes induced by sequentially consistent (SC) executions into two disjoint sets:

$$Explored \sqcup Unexplored$$

For clarity, note that *Explored* here refers to this partition, and is not the same as the algorithm's internal set EXPLORED.

We say that an execution belongs to *Explored* (respectively, *Unexplored*) if it belongs to a view equivalence class contained in that set.

Assume, for contradiction, that the algorithm terminates while $Unexplored \neq \emptyset$. Let E be an execution in *Unexplored*.

Lemma 4.1. *There exists a prefix $E'' = E'e_r$ of E (e_r a read operation, E' possibly empty) such that both:*

1. *The read-cut induced by E' is a read-cut of some execution in Explored.*
2. *The read-cut induced by E'' is not a read-cut of any execution in Explored.*

Proof. Suppose, for contradiction, that no such prefix exists. Then either:

1. Every prefix E' of E induces a read-cut not induced by any execution in *Explored*. This is impossible, since the empty prefix induces the empty read-cut, which is common to all executions in *Explored*. (Note that *Explored* is non-empty, because the algorithm always runs at least one execution at initialization.)
2. Every prefix $E'' = E'e_r$ of E induces a read-cut that is also induced by some execution in *Explored*. In particular, let $E'' = E$. Then E induces the same read-cut as some $\widetilde{E} \in Explored$. Hence E and $\widetilde{E}$ belong to the same view equivalence class, contradicting the assumption that $E \in Unexplored$ while $\widetilde{E} \in Explored$.

This contradiction proves the lemma. □

Let $E'' = E'e_r$ be the prefix guaranteed by the lemma. Let $\widetilde{E} \in Explored$ be an execution that has been run such that $rc_{E'}$ is a read-cut of $\widetilde{E}$.[4]

Since $rc_{E'}$ is a read-cut of both E and $\widetilde{E}$, and since $\widetilde{E}$ has run, the algorithm eventually examines $rc_{E'}$, as it explores all read-cuts of $\widetilde{E}$.

The difference between $rc_{E'}$ and $rc_{E''}$ is exactly one additional read: e_r.

In E, the read e_r reads-from some write e_w in E'. It follows that e_w is contained in

> the maximal prefix of $E_{e_w.\texttt{thread}}$ containing exactly $|rc_{E'}(e_w.\texttt{thread})|$ reads,

and therefore, by thread determinism, and since $rc_{E'}$ is a read-cut of both E and $\widetilde{E}$, also in

> the maximal prefix of $\widetilde{E}_{e_w.\texttt{thread}}$ containing exactly $|rc_{E'}(e_w.\texttt{thread})|$ reads.

Hence, when the exploration algorithm checks for new possible values for e_r, $e_w.\texttt{value} \in$ VALUES. Therefore, the expanded read-cut $rc_{E''}$ induced by E'' is indeed examined by the algorithm.

Moreover, by thread determinism, the $rc_{E'}$-restrictions of E and $\widetilde{E}$ (namely $E_{rc_{E'}} = E'$ and $\widetilde{E}_{rc_{E'}}$) consist of the same operations, differing only in the interleaving across threads.[5]

Thus, the sequence constructed by the algorithm as input to CONSTRUCTWITNESS is, as a set of operations, precisely $E'e_r$. Since $E'e_r = E''$ is a feasible execution prefix, CONSTRUCTWITNESS returns an execution prefix rather than NONE. The algorithm then invokes the oracle SCHEDULER on this prefix.

By the construction of CONSTRUCTWITNESS, the execution that now runs has the read-cut $rc_{E''}$, and having run, it belongs to $Explored$. This contradicts the lemma's assumption that $rc_{E''}$ is not a read-cut of any execution in $Explored$.

This contradiction completes the proof. □

4.6 Reducing Memory Consumption

At an abstract level, the exploration algorithm operates as a **graph traversal**.

Each node in this graph represents a *view equivalence class*, and a directed edge connects two nodes $C_1 \to C_2$ whenever a feasible read-cut of C_1 can be extended into a feasible read-cut of C_2 by a single read expansion. Importantly, these edges are discovered *on the fly*: they are generated only when the source node is visited. Moreover, upon visiting a node, the algorithm computes *all* of its outgoing edges immediately, prior to exploring any other node. Consequently, these edges must be stored in order to enable further exploration along them at a later stage.

[4] Such an execution $\widetilde{E}$ must exist because:

1. $rc_{E'}$ is a read-cut of a view equivalence class that belongs to $Explored$,
2. all executions within the same view equivalence class have the same read-cuts, and
3. since this view equivalence class is in $Explored$, at least one execution from it has been run by the algorithm.

[5] For every thread $p \in dom(rc_{E'})$:

1. the maximal prefix of E_p containing exactly $|rc_{E'}(p)|$ reads, and
2. the maximal prefix of $\widetilde{E}_p$ containing exactly $|rc_{E'}(p)|$ reads

are identical. By thread determinism, these prefixes must be equal. Hence $E_{rc_{E'}} = \widetilde{E}_{rc_{E'}}$.

The graph is never constructed explicitly. Instead, the traversal is driven by the container Executions, which stores executions (nodes) to be explored. Soundness and optimality require only that Executions behaves as a generic container: executions are inserted into it and later extracted, in any order. Correctness does not depend on the extraction order, and uniqueness of entries need not be enforced.

However, the order in which executions are extracted – i.e., the traversal strategy – has a major effect on *memory consumption*. Each time a node is visited, its neighboring nodes are generated and temporarily stored in Executions for later exploration. Thus, the traversal strategy determines the maximum number of nodes stored at once.

If Executions is implemented as a *stack* (depth-first traversal), the container may grow large: upon reaching the leftmost leaf, the stack holds the entire traversal tree, except for the nodes along the current root-to-leaf path.

Conversely, if Executions is a *queue* (breadth-first traversal), the container typically holds only one or two layers of the traversal tree at a time.

Neither traversal tree need resemble a balanced tree: they may degenerate into long chains. However, empirical results on real benchmarks suggest the following pattern: the upper levels of the traversal tree exhibit high fanout, while fanout decreases near the leaves. Equivalently, the graph of view equivalence classes resembles a dense, highly connected core with chain-like structures extending from it.

Under this structure, the number of nodes in one or two breadths of the BFS tree is typically much smaller than the number of nodes in the DFS tree minus a single root-to-leaf path. We therefore expect BFS traversal to require significantly less memory.

Experiments comparing DFS and BFS confirm this expectation. Consequently, we adopt BFS traversal in practice, using a queue as the data structure for Executions.

A final caveat is worth noting. This discussion has implicitly assumed that all edges of the view equivalence class graph require approximately the same computational effort to explore. This need not hold: for certain edges, the heuristics of ConstructWitness may fail, causing those edges to require disproportionately more time. If such edges become part of the traversal tree, the time to visit them may dominate the exploration cost. It is conceivable that some traversal strategies make such edges more or less likely to appear. However, in our experiments we have not observed such an effect.

4.7 Improved Read-Cut Enumeration with Early Feasibility Pruning

Recall that the set of read-cuts of an execution E was earlier defined as the cross-product

$$\prod_{p \in \mathsf{Threads}(E)} \mathrm{Pref}(rc_E(p)),$$

While conceptually simple, this naive enumeration generates an exponential number of candidates, most of which are trivially infeasible: they prescribe that some read must return a value that is never produced by any write in the corresponding rc-restriction of E.

To avoid this blow-up, we design a specialised iterator that enumerates read-cuts *incrementally* while simultaneously checking a necessary feasibility condition. The key idea is to maintain the sets of the

values written and the values read so far. If a read-cut requires a value v for which no write exists in the current rc-restriction, the candidate is immediately discarded, and enumeration continues with the next candidate. Thus, many infeasible read-cuts are pruned on the fly, without ever being materialised.

The enumeration itself is structured as a form of mixed-radix counting across threads: each thread p is treated as a digit whose value corresponds to the number of reads included from p. Incrementing a digit corresponds to extending the prefix of p by one additional read, while backtracking (carrying to the left) resets a prefix and removes the contributions of its reads and writes. Throughout, the following three auxiliary structures are maintained and updated incrementally:

- W_{rc}: multiset of values written by all write operations that currently belong to the rc-restriction E_{rc}.
- R_{rc}: multiset of values read by the reads currently included in rc.
- Unsat_{rc}, the set of *unsatisfied* read values, i.e. those required by R_{rc} but not yet provided by W_{rc}. Formally, $\mathsf{Unsat} = \mathsf{R}_{rc} \setminus \mathsf{W}_{rc}$.

  Note that, for this definition, W_{rc} and R_{rc} are interpreted as sets rather than multisets.

A read-cut is emitted only when $\mathsf{Unsat} = \emptyset$. This ensures that every enumerated read-cut satisfies the necessary condition that all of its reads can be matched to at least one write in its restriction E_{rc}.

Algorithm 2: EnumerateReadCutsWithPruning(E)

Input: Execution E

Output: Stream of read-cuts rc that satisfy a necessary feasibility condition.

$rc \leftarrow \emptyset$ `// all thread prefixes empty`
$\mathsf{R}_{rc} \leftarrow \emptyset$
$\mathsf{Unsat} \leftarrow \emptyset$
$\mathsf{W}_{rc} \leftarrow$ multiset of values written by all writes occurring before the first read of each thread p in E_p
yield rc
while *true* **do**
  $i \leftarrow |\mathsf{Threads}(E)|$
  while $i \geq 1$ **and** *thread p (the i-th thread) has no further read to include* **do**
    let p be the i-th thread. `// Reset (carry out) the digit of` p`: undo its current contribution to` E_{rc}
    foreach *read e_r of E_p* **do**
      `// Remove reads of` p
      $v \leftarrow e_r$.`value`
      decrement $\mathsf{R}_{rc}(v)$
      if $\mathsf{R}_{rc}(v) > 0$ **and** $\mathsf{W}_{rc}(v) = 0$ **then** insert v into Unsat
      if $\mathsf{R}_{rc}(v) = 0$ **then** remove v from Unsat
    foreach *write e_w of E_p that is after the first read of E_p* **do**
      `// Remove writes of` p
      $v \leftarrow e_w$.`value`
      decrement $\mathsf{W}_{rc}(v)$
      if $\mathsf{R}_{rc}(v) > 0$ **and** $\mathsf{W}_{rc}(v) = 0$ **then** insert v into Unsat
    $rc(p) \leftarrow []$
    $i \leftarrow i - 1$
  if $i < 1$ **then break** `// Overflowed the most-significant digit`
  `// Increment the digit of` p`: include its next read`
  let e_r be the next read of p in E_p
  $v \leftarrow e_r$.`value`
  append v to $rc(p)$
  increment $\mathsf{R}_{rc}(v)$
  if $\mathsf{W}_{rc}(v) = 0$ **then** insert v into Unsat
  foreach *write e_w of E_p occurring after e_r and before the next read of p (or end of E_p)* **do**
    `// Expand` E_{rc} `along` p`: include writes up to (but not including) the next read of` p
    $w \leftarrow e_w$.`value`
    increment $\mathsf{W}_{rc}(w)$
    if $w \in \mathsf{Unsat}$ **and** $\mathsf{W}_{rc}(w) > 0$ **then** remove w from Unsat
  if $\mathsf{Unsat} = \emptyset$ **then yield** rc

Once defined, EnumerateReadCutsWithPruning replaces the naive cross-product enumeration in the exploration algorithm. Concretely, wherever the baseline algorithm iterates

$$\text{for all read-cuts } rc \text{ of an execution } E$$

we instead iterate

$$\text{for all read-cuts } rc \in \textsc{EnumerateReadCutsWithPruning}(E)$$

This refinement preserves completeness while avoiding the explosion of trivially infeasible read-cuts. By pruning early, the exploration algorithm focuses its effort only on potentially feasible cases, substantially reducing the number of candidate read-cuts that must later be checked by the more costly consistency algorithms.

A crucial consequence is that trivially infeasible read-cuts are never inserted into Explored or ToBeExplored. This yields a *substantial reduction in memory consumption*, as these sets would otherwise accumulate many read-cuts that cannot correspond to any sequentially consistent execution.

In principle, one might attempt to avoid storing all infeasible read-cuts. However, doing so would force the algorithm to re-examine the same infeasible read-cuts across different executions, incurring significant overhead due to repeatedly invoking ConstructWitness on read-cuts that have already been determined to be infeasible. In contrast, our enumeration algorithm discards trivially infeasible read-cuts immediately: their infeasibility is detected in constant time during enumeration, and because every candidate is traversed regardless, there is no advantage in storing them.

5 The Algorithm ConstructWitness

We now describe ConstructWitness, the algorithm that, given a sequence of operations E, attempts to build a reordering $\widetilde{E}$ that is a prefix of some sequentially consistent (SC) execution - or prove that none exists. Our presentation is modular. First, we give a complete algorithm (Section 5.1) that reduces the problem to a constraint-solving problem solved by an SMT solver; this serves as the complete fallback. Then we introduce two polynomial-time heuristics that dramatically reduce solver usage in practice: Heuristic-URF (Section 5.2), which infers a mandatory subset of the happens-before relation and quickly rejects many infeasible cases, and Heuristic-INCR (Section 5.3), which incrementally assembles SC execution prefixes by reusing relations from the most recent execution while accommodating a newly added read. The final algorithm (Section 5.4) combines these components: It first runs Heuristic-URF. If it proves the candidate is Inconsistent, it stops. Otherwise it invokes Heuristic-INCR to try to construct a witness. Only if Heuristic-INCR returns Unknown does it fall back to ConstructWitnessComplete.

5.1 ConstructWitnessComplete: A Complete Algorithm based on Constraint-Solving

We define an algorithm for checking whether there exists a reordering $\widetilde{E}$ of a given sequence of operations E such that $\widetilde{E}$ is a prefix of some sequentially consistent (SC) execution.

The problem is expressed as a constraint satisfaction problem in *quantifier-free integer difference logic* (QF-IDL)[6]. Unlike standard integer programming, our encoding requires disjunctions (logical OR), which are not directly supported by vanilla integer programming but are handled by SMT solvers such as Z3.

Theorem 5.1 (Necessary and Sufficient Conditions for ConstructWitnessComplete)**.** *A sequence of operations $\widetilde{E}$ is a prefix of an SC execution if and only if program order is respected and, for every read operation $e_r \in \widetilde{E}$, the following conditions hold:*

1. *There exists a write e_w to e_r.*`address` *with value e_r.*`value` *such that e_w occurs before e_r in $\widetilde{E}$.*
2. *Every other write $e_w^\star \neq e_w$ to e_r.*`address` *occurs either strictly before e_w or strictly after e_r.*

Note that even other writes to the same memory object with the same value are disallowed between e_w and e_r.

Proof. ($\Rightarrow$) If $\widetilde{E}$ is a prefix of an SC execution, then (i) program order is preserved by definition of SC, and (ii) every read e_r reads the value of the latest preceding write to e_r.`address`; hence Conditions (1)-(2) hold.

($\Leftarrow$) Assume program order is preserved and Conditions (1)-(2) hold for every read $e_r \in \widetilde{E}$. $e_1 \xrightarrow{\widetilde{E}} e_2$ denotes that e_1 occurs before e_2 in the sequence $\widetilde{E}$.

Define the relations.

[6]Quantifier-Free Integer Difference Logic (QF-IDL) consists of atomic constraints of the form $x - y \leq c$, x,y are integer variables and c is an integer constant, combined using the standard Boolean connectives. The satisfiability problem for QF-IDL is decidable; for instance, it can be solved within the DPLL(T) framework [23].

Program order $\mathtt{po}$**.** For each thread p, define $e_1 \xrightarrow{\mathtt{po}} e_2$ iff e_1, e_2 belong to p and $e_1 \xrightarrow{\widetilde{E}} e_2$. By assumption, program order is respected in $\widetilde{E}$, so $\mathtt{po} \subseteq \xrightarrow{\widetilde{E}}$.

Reads-from $\mathtt{rf}$**.** For each read e_r, Conditions (1)–(2) ensure that there exists a write e_w to $e_r.\mathtt{address}$ with value $e_r.\mathtt{value}$ such that e_w occurs before e_r, and moreover that e_w is the latest such write. Define $e_w \xrightarrow{\mathtt{rf}} e_r$. Since $e_w \xrightarrow{\widetilde{E}} e_r$, it follows that $\mathtt{rf} \subseteq \xrightarrow{\widetilde{E}}$.

Coherence $\mathtt{co}$**.** For each memory object $\mathtt{address}$, define $e_{w1} \xrightarrow{\mathtt{co}} e_{w2}$ for writes to $\mathtt{address}$ iff $e_{w1} \xrightarrow{\widetilde{E}} e_{w2}$. Thus, $\mathtt{co}$ is acyclic, total per address, and $\mathtt{co} \subseteq \xrightarrow{\widetilde{E}}$ by construction.

Handling $\mathtt{rb}$*.* Let $\mathtt{rb} := \mathtt{rf}^{-1}; \mathtt{co}$. We show $\mathtt{rb} \subseteq \xrightarrow{\widetilde{E}}$. Pick any edge $e_r \xrightarrow{\mathtt{rb}} e'_w$. Then there exists a write e_w such that $e_w \xrightarrow{\mathtt{rf}} e_r$ and $e_w \xrightarrow{\mathtt{co}} e'_w$. From $e_w \xrightarrow{\mathtt{co}} e'_w$ and the current construction of $\mathtt{co}$, we have $e_w \xrightarrow{\widetilde{E}} e'_w$.

Apply Condition (2) of the theorem to the read e_r: every write $e^\star \neq e_w$ to $e_r.\mathtt{address}$ occurs either strictly before e_w or strictly after e_r in $\widetilde{E}$. Instantiate $e^\star := e'_w$. Since $e_w \xrightarrow{\widetilde{E}} e'_w$, e'_w cannot be strictly before e_w; therefore $e_r \xrightarrow{\widetilde{E}} e'_w$, and thus $\mathtt{rb} \subseteq \xrightarrow{\widetilde{E}}$.

Acyclicity and conclusion. We have shown $\mathtt{po}, \mathtt{rf}, \mathtt{co}, \mathtt{rb} \subseteq \xrightarrow{\widetilde{E}}$, so $\mathtt{hb} := \mathtt{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{rb} \subseteq \xrightarrow{\widetilde{E}}$. Since $\xrightarrow{\widetilde{E}}$ is acyclic, so is $\mathtt{hb}$. Moreover, $\widetilde{E}$ is a topological sort of $\xrightarrow{\widetilde{E}}$, and therefore also of $\mathtt{hb}$.

By the SC Acyclicity Theorem (Theorem 2.1), it follows that $\widetilde{E}$ is a prefix of some SC execution. □

We introduce an integer variable V_e for each operation e, representing its position in the reordered sequence $\widetilde{E}$.

The constraints are expressed in QF-IDL as follows.

Range and Distinctness. To ensure that each V_e corresponds to a valid position in $\widetilde{E}$ and thus that $\widetilde{E}$ defines a valid total order, the V_e must lie within the bounds of the execution and be pairwise distinct:

$$\mathsf{Valid} := \forall e \in E : \ 0 \leq V_e < |E| \quad \wedge \quad \text{all } V_e \text{ are distinct}$$

Program Order Preservation. For each thread p, the reordering must respect the relative order of its operations in E:

$$\mathsf{PO} := \bigwedge_{p \in \mathsf{Threads}(E)} \ \bigwedge_{(e_i, e_j) \text{ consecutive in } E_p} V_{e_i} < V_{e_j} \tag{1}$$

Reads-from Condition. For each read e_r, there must exist some write e_w to the same address with value $e_r.\mathtt{value}$ such that:

1. e_w occurs before e_r, and
2. every other write $e'_w \neq e_w$ to the same address occurs either strictly before e_w or strictly after e_r.

Let:

$$\mathcal{W}(\mathtt{address}, v) = \{e_w \in E \mid e_w \text{ is a write}, e_w.\mathtt{address} = \mathtt{address}, e_w.\mathtt{value} = v\},$$

$$\mathcal{W}(\texttt{address}) = \{e_w \in E \mid e_w \text{ is a write, } e_w.\texttt{address} = \texttt{address}\}.$$

The constraint for e_r is:

$$\mathsf{R}(e_r) := (\mathcal{W}(e_r.\texttt{address}, e_r.\texttt{value}) \neq \emptyset) \wedge \bigvee_{e_w \in \mathcal{W}(e_r.\texttt{address}, e_r.\texttt{value})} \Big(V_{e_w} < V_{e_r} \ \wedge \bigwedge_{\substack{e'_w \in \mathcal{W}(e_r.\texttt{address}) \\ e'_w \neq e_w}} (V_{e'_w} < V_{e_w} \ \vee \ V_{e_r} < V_{e'_w}) \Big). \tag{2}$$

Final Constraint. The overall formula is:

$$\mathbf{P} := \mathsf{Valid} \wedge \mathsf{PO} \wedge \bigwedge_{\substack{e_r \in E \\ e_r \text{ is a read}}} \mathsf{R}(e_r). \tag{3}$$

The formula $\mathbf{P}$ is in QF-IDL, using only atomic constraints of the form $x < y$. Since QF-IDL is decidable [23], satisfiability can be checked effectively with an SMT solver such as Z3.

Finally, the pseudocode of ConstructWitnessComplete is given in Algorithm 3.

Algorithm 3: ConstructWitnessComplete(E)

```
Construct formula P
Invoke an SMT solver on P
if Unsatisfiable then return None
else
    Extract model and order operations of E by V_e to obtain Ẽ
    return Ẽ
```

5.2 Heuristic-URF: Fast Elimination of Inconsistent Executions

The majority of invocations of ConstructWitness are expected to fail, i.e. most candidate read-cuts are infeasible and do not correspond to any prefix of a sequentially consistent execution. To avoid the overhead of repeatedly invoking an SMT solver, we introduce a polynomial-time heuristic that eliminates a large fraction of these infeasible cases. The heuristic returns either Inconsistent, indicating that no valid execution exists, or Unknown.

The heuristic is motivated by [5], which, though, operates under the assumption that the entire reads-from relation is known. In our setting, however, the reads-from relation is not specified.

We introduce a weaker relation than `rf`, termed *Unique Reads-From* (`urf`), which is a subrelation of `rf`.

Given a sequence of operations E, the Heuristic-URF algorithm incrementally constructs a partial happens-before relation `hb`. At each step of the algorithm, `hb` contains only those edges that must necessarily hold in any sequentially consistent execution prefix that is a reordering of E. This partial relation is then used to infer mandatory reads-from edges, which we term as *Unique Reads-From* (`urf`).

Definition 5.1 (Unique Reads-From (URF))**.** *Let E be a sequence of operations, and let* `hb` *denote the current partial happens-before relation maintained by the algorithm. A read operation $e_r \in E$ is*

said to uniquely read-from *the write operation* $e_w \in E$*, denoted* $e_w \xrightarrow{\texttt{urf}} e_r$*, if and only if:*

1. $e_w.\texttt{address} = e_r.\texttt{address}$ *and* $e_w.\texttt{value} = e_r.\texttt{value}$*, and*
2. *There does not exist any write* $e'_w \neq e_w$ *such that* $e'_w \in \mathcal{W}(e_r.\texttt{address}, e_r.\texttt{value})$ *and* $e'_w \not\xrightarrow{\texttt{hb}} e_r$.

If $e_w \xrightarrow{\texttt{urf}} e_r$, then $e_w \xrightarrow{\texttt{rf}} e_r$ in every sequentially consistent execution prefix that is a reordering of E.

On the basis of this definition, the HEURISTIC-URF algorithm proceeds by iteratively extending the partial happens-before relation hb until a fixpoint is reached. At each iteration, urf edges are inferred and, along with their consequences, are added to hb. If this process yields a cycle, the candidate execution is declared INCONSISTENT. Otherwise, the algorithm continues until a fixpoint is reached.

Instead of performing explicit cycle detection, each time we add an edge $e_1 \xrightarrow{\texttt{hb}} e_2$, we first check whether $e_2 \xrightarrow{\texttt{hb}} e_1$ already holds. If it does, we return INCONSISTENT.

The algorithm is defined as follows:

1. Initialise hb with program-order po.
2. Iterate to a fixpoint:
   (a) urf **inference.** For each read $e_r \in E$ with exactly one candidate write $e_w \in \mathcal{W}(e_r.\texttt{address}, e_r.\texttt{value})$ such that $e_r \not\xrightarrow{\texttt{hb}} e_w$, add the edge $e_w \xrightarrow{\texttt{urf}} e_r$. If no such e_w exists, return INCONSISTENT.

   (b) co **inference.** For each edge $e_w \xrightarrow{urf} e_r$ and for all $e'_w \in \mathcal{W}(e_w.\texttt{address})$ with $e'_w \xrightarrow{\texttt{hb}} e_r$, add $e'_w \xrightarrow{\texttt{co}} e_w$ (recorded as $e'_w \xrightarrow{\texttt{hb}} e_w$). This must hold because if, by contradiction, $e_w \xrightarrow{\texttt{co}} e'_w$, then
   $$e_w \xrightarrow{\texttt{co}} e'_w \xrightarrow{\texttt{hb}} e_r,$$
   implying e_r does not read-from e_w since e'_w overwrites $e_w.\texttt{address}$ – an absurdity.

   (c) rb **computation.** For each edge $e_w \xrightarrow{\texttt{urf}} e_r$ and for all $e'_w \in \mathcal{W}(e_w.\texttt{address})$ such that $e_w \xrightarrow{\texttt{hb}} e'_w$, add $e_r \xrightarrow{\texttt{rb}} e'_w$. Recall that $\texttt{rb} = \texttt{rf}^{-1}; \texttt{co}$.

   (d) If a cycle is detected in hb, return INCONSISTENT.
3. On reaching a fixpoint without inconsistency, return (urf, hb).

Rules (2) and (3) coincide with the saturation rules of [5], but here they are reformulated directly in terms of the relations used by the SC Acyclicity Theorem (Theorem 2.1).

At first glance, it may seem that co **inference** and rb **computation** computation need only consider newly added urf edges in each iteration. However, as the algorithm progresses, hb accumulates more constraints, and inferences may become possible for older urf edges as well. Hence iterating over urf at rb **computation** and co **inference** is necessary.

The pseudocode of HEURISTIC-URF is given in Algorithm 4.

Algorithm 4: HEURISTIC-URF(E)

Input: Sequence of operations E

Output: INCONSISTENT or $(\texttt{urf}, \texttt{hb})$

$\texttt{urf} \leftarrow \emptyset$

$\texttt{hb} \leftarrow \emptyset$

foreach *thread* $p \in \mathsf{Threads}(E)$ **do**

  `// Initialise hb with program-order edges.`

  $E_p \leftarrow [\, e \in E \mid e.\texttt{thread} = p \,]$

  foreach *consecutive* (e_1, e_2) *in* E_p **do**

    Add $e_1 \xrightarrow{\text{po}} e_2$ to hb

while *true* **do**

  $\mathit{progress} \leftarrow$ **false**

  foreach *read* $e_r \in E$ *with no incoming* urf *edge* **do** `// --- urf inference ---`

    $\mathcal{W} \leftarrow \{\, e_w \mid e_w \in \mathcal{W}(e_r.\texttt{address}, e_r.\texttt{value}) \wedge e_r \not\xrightarrow{\text{hb}} e_w \,\}$

    if $\mathcal{W} = \emptyset$ **then return** INCONSISTENT

    if $|\mathcal{W}| = 1$ **then**

      $\mathit{progress} \leftarrow$ **true**

      let $\mathcal{W} = \{e_w\}$

      Add $e_w \xrightarrow{\text{urf}} e_r$ to urf and to hb

  foreach *edge* $(e_w, e_r) \in \texttt{urf}$ **do** `// --- co inference ---`

    foreach *write* $e'_w \in \mathcal{W}(e_w.\texttt{address})$ *with* $e'_w \xrightarrow{\text{hb}} e_r$ **do**

      $\mathit{progress} \leftarrow$ **true**

      if $e_w \xrightarrow{\text{hb}} e'_w$ **then return** INCONSISTENT `// Cycle Detection`

      Add $e'_w \xrightarrow{\text{co}} e_w$ to hb

  foreach *edge* $(e_w, e_r) \in \texttt{urf}$ **do** `// --- rb computation ---`

    foreach $e'_w \in \mathcal{W}(e_w.\texttt{address})$ *with* $e_w \xrightarrow{\text{hb}} e'_w$ **do**

      $\mathit{progress} \leftarrow$ **true**

      if $e'_w \xrightarrow{\text{hb}} e_r$ **then return** INCONSISTENT `// Cycle Detection`

      Add $e_r \xrightarrow{\text{rb}} e'_w$ to hb

  if ***not*** *progress* **then break**

return $(\texttt{urf}, \texttt{hb})$

Because the algorithm makes many connectivity queries on hb, we maintain hb as its transitive closure. This allows reachability checks in $O(1)$ by simple edge lookup. To preserve transitivity, whenever adding an edge (a, b), the following updates are also made:

- For all x, y with $(x, a), (b, y) \in \texttt{hb}$, add (x, y).
- For all x with $(x, a) \in \texttt{hb}$, add (x, b).
- For all y with $(b, y) \in \texttt{hb}$, add (a, y).

The heuristic is sound: every sequence of operations reported as INCONSISTENT is indeed infeasible under Sequential Consistency. Because the problem of deciding execution consistency under Sequential Consistency is NP-complete, no polynomial-time algorithm can be complete (unless P = NP). Accordingly, the heuristic must sacrifice completeness: some cases remain unresolved and are delegated to the complete, but more costly, algorithm based on constraint-solving.

In practice, however, the heuristic rapidly eliminates nearly all inconsistent candidate executions, leaving only a small fraction to be resolved by CONSTRUCTWITNESSCOMPLETE.

5.3 HEURISTIC-INCR: Incremental Construction of Sequentially Consistent Executions

HEURISTIC-URF substantially reduces the computational burden of rejecting infeasible read-cuts. However, whenever a read-cut is feasible and an actual sequentially consistent (SC) execution prefix must be constructed, CONSTRUCTWITNESSCOMPLETE is required, which significantly impacts the overall performance of the SMC algorithm.

To mitigate this cost, we introduce HEURISTIC-INCR, a polynomial-time heuristic that attempts to incrementally construct sequentially consistent execution prefixes by reusing information from previously run sequentially consistent executions.

Before HEURISTIC-INCR is invoked, the following are already available:

- A *partial* reads-from relation $\mathtt{rf}$ (specifically, the $\mathtt{urf}$ subset) and a *partial* happens-before relation $\mathtt{hb}$, both guaranteed to be subsets of the corresponding relations of any SC execution prefix that is a reordering of E.
- The most recent complete SC execution E_{pre}, from which the current sequence E is obtained as a subsequence of E_{pre} extended by one new read operation.

Two elements are missing in order to obtain an SC execution prefix for E:

1. A *complete* reads-from relation.
2. A *complete* coherence order (*total* order on writes to the same memory object).

The heuristic attempts to complete the $\mathtt{rf}$ and $\mathtt{co}$ relations by reusing $\mathtt{rf}_{\mathsf{pre}}$ and $\mathtt{co}_{\mathsf{pre}}$. Since E is obtained by minimally modifying E_{pre}, these relations are expected to remain valid for most operations. If successful, the resulting $\mathtt{hb} = \mathtt{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{rb}$ will be acyclic, and a sequentially consistent execution prefix $\widetilde{E}$ can be obtained by a topological sort of $\mathtt{hb}$. If at any point a cycle is introduced to $\mathtt{hb}$, the heuristic cannot conclude infeasibility, and defers the decision to the complete algorithm based on constraint solving.

Unlike CONSTRUCTWITNESSCOMPLETE and HEURISTIC-URF, which only take the sequence E as input, HEURISTIC-INCR additionally receives the new read operation e_r^{new} introduced by the exploration algorithm. e_r^{new} is the only operation not present in E_{pre}, and thus the only one for which we have no information beyond the partial relations inferred by HEURISTIC-URF. The heuristic tries all candidate writes as a read-from write for e_r^{new}.

The heuristic requires access to the relations $\mathtt{rf}_{\mathsf{pre}}$ and $\mathtt{co}_{\mathsf{pre}}$ of E_{pre}. The input E is a subsequence of E_{pre} with the additional read e_r^{new}. Therefore, the restrictions of $\mathtt{rf}_{\mathsf{pre}}$ and $\mathtt{co}_{\mathsf{pre}}$ to the operations of

E coincide with the relations obtained by computing $\texttt{rf}$ and $\texttt{co}$ directly from E and then discarding any edges involving e_r^{new}.

Choosing a Candidate Source Write. For each write $e_w^{\text{new}} \in \mathcal{W}(e_r^{\text{new}}.\texttt{address}, e_r^{\text{new}}.\texttt{value})$ (defined with respect to E) such that $e_r \overset{\texttt{hb}}{\not\longrightarrow} e_w^{\text{new}}$, attempt the following construction. The first successful attempt yields an SC prefix. If none succeeds, return UNKNOWN.

Completing the $\texttt{rf}$ Relation.

- Initialize $\texttt{rf} := \texttt{urf}$, add $e_w^{\text{new}} \overset{\texttt{rf}}{\longrightarrow} e_r^{\text{new}}$ to $\texttt{rf}$ and to $\texttt{hb}$.
- For each read $e_r \in E$ without a source write in $\texttt{rf}$:
  1. First try the source e_w such that $e_w \overset{\texttt{rf}_{\text{pre}}}{\longrightarrow} e_r$. If $e_w \in E$ [7] and $e_r \overset{\texttt{hb}}{\not\longrightarrow} e_w$, add $e_w \overset{\texttt{rf}}{\longrightarrow} e_r$ to $\texttt{rf}$ and $\texttt{hb}$.
  2. Otherwise, search for an alternative write operation as follows. Starting from the position of e_r in the sequence E, scan backwards towards the beginning of E. At each step, if the encountered operation e_w is a write such that $e_w.\texttt{address} = e_r.\texttt{address}$, $e_w.\texttt{value} = e_r.\texttt{value}$, and $e_r \overset{\texttt{hb}}{\not\longrightarrow} e_w$, select this e_w as the source write of e_r by adding the edge $e_w \overset{\texttt{rf}}{\longrightarrow} e_r$ to $\texttt{rf}$ and to $\texttt{hb}$.

     If no such write is found in the backward scan, repeat the procedure by scanning forwards from the position of e_r towards the end of E. If no suitable write is found in either direction, return UNKNOWN.

     The rationale for this search order is twofold. First, the backward scan prioritises writes already ordered before e_r in the given sequence, which are more likely to respect existing constraints. Second, examining operations close to e_r increases the chance of success, as nearby candidates typically introduce fewer additional happens-before dependencies.
  3. Apply $\texttt{co}$ inference and $\texttt{rb}$ computation (as in HEURISTIC-URF).

This step ensures $\texttt{rf}$ is complete, as required by the SC Acyclicity Theorem (Theorem 2.1).

Completing the $\texttt{co}$ relation. At this stage, some pairs of writes to the same memory object may remain unordered in $\texttt{hb}$. To complete the coherence order, we reuse information from the previously known execution E_{pre}.

For each memory object $\texttt{address} \in dom(\mathcal{W})$, consider all pairs of distinct writes $e_{w1}, e_{w2} \in \mathcal{W}(\texttt{address})$. If neither $e_{w1} \overset{\texttt{hb}}{\longrightarrow} e_{w2}$ nor $e_{w2} \overset{\texttt{hb}}{\longrightarrow} e_{w1}$ holds, then inherit their order from $\texttt{hb}_{\text{pre}}$:

- If $e_{w1} \overset{\texttt{hb}_{\text{pre}}}{\longrightarrow} e_{w2}$, add the edge $e_{w1} \overset{\texttt{co}}{\longrightarrow} e_{w2}$ to $\texttt{hb}$.
- If $e_{w2} \overset{\texttt{hb}_{\text{pre}}}{\longrightarrow} e_{w1}$, add the edge $e_{w2} \overset{\texttt{co}}{\longrightarrow} e_{w1}$ to $\texttt{hb}$.

Recall that the $\texttt{co}$ relation is not maintained explicitly; instead, its edges are incorporated directly into $\texttt{hb}$.

[7] Because we compute $\texttt{rf}_{\text{pre}}$ from E, for this case, we will just not find a source write for read e_r in the computed $\texttt{rf}_{\text{pre}}$.

`rb` computation. Finally, add all edges $\texttt{rb} = \texttt{rf}^{-1}; \texttt{co}$ to `hb`. If this introduces a cycle, return UNKNOWN.

Topological Sort. At this stage, we have a complete reads-from relation `rf` and a total coherence order `co` for all writes, together with program order `po` and the derived relation `rb`. We also have that the derived relation `hb` from these relations is acyclic. These relations satisfy the requirements of the SC Acyclicity Theorem (Theorem 2.1).

Accordingly, perform a topological sort of `hb`. The resulting sequence $\widetilde{E}$ is, by the SC Acyclicity Theorem (Theorem 2.1), a sequentially consistent execution prefix. Return $\widetilde{E}$.

The pseudocode of HEURISTIC-INCR is given in Algorithm 5.

Algorithm 5: HEURISTIC-INCR($E, e_r^{\text{new}}, (\texttt{urf}, \texttt{hb}_0)$)

Input: Sequence of operations E, new read e_r^{new}, $(\texttt{urf}, \texttt{hb}_0)$ from HEURISTIC-URF

Output: UNKNOWN or SC execution prefix $\widetilde{E}$

Group writes of E by $(\texttt{address}, \texttt{value})$ into the sets $\mathcal{W}(\texttt{address}, \texttt{value})$.

$\texttt{rf}_{\text{pre}}, \texttt{co}_{\text{pre}} \leftarrow$ compute rf, co from E, omitting all edges involving e_r^{new}

`// These equal the restrictions of the relations of` E_{pre} `to the operations in` E.

foreach *write* $e_w^{\text{new}} \in \mathcal{W}(e_r^{\text{new}}.\texttt{address}, e_r^{\text{new}}.\texttt{value})$ *such that* $e_r^{\text{new}} \overset{\texttt{hb}_0}{\not\longrightarrow} e_w^{\text{new}}$ **do**

  `// --- choosing a candidate source write for the new read` e_r^{new} `---`

  $\texttt{hb} \leftarrow \texttt{hb}_0$; $\texttt{rf} \leftarrow \texttt{urf}$

  Add $e_w^{\text{new}} \overset{\texttt{rf}}{\longrightarrow} e_r^{\text{new}}$ to rf and to hb

  foreach *read* $e_r \in E$ **do** `// --- completing the rf relation ---`

    if $\nexists e_w$ *with* $e_w \overset{\texttt{rf}}{\longrightarrow} e_r$ **then**

      $e_w \leftarrow$ the write such that $e_w \overset{\texttt{rf}_{\text{pre}}}{\longrightarrow} e_r$

      if $e_w \neq$ NONE *(this can occur if* $e_w \in E_{\text{pre}}$ *but* $e_w \notin E$*)* ***and*** $e_r \overset{\texttt{hb}}{\not\longrightarrow} e_w$ **then**

        Add $e_w \overset{\texttt{rf}}{\longrightarrow} e_r$ to rf and to hb

      else `// --- search for a suitable alternative source write for` e_r `---`

        $e_w \leftarrow$ first write found when scanning *backwards* in E from e_r toward the start that satisfies: $e_w.\texttt{address} = e_r.\texttt{address} \wedge e_w.\texttt{value} = e_r.\texttt{value} \wedge e_r \overset{\texttt{hb}}{\not\longrightarrow} e_w$ (NONE if none exists)

        if e_w *is* NONE **then**

          $e_w \leftarrow$ first write found when scanning *forwards* in E from e_r toward the end that satisfies: $e_w.\texttt{address} = e_r.\texttt{address} \wedge e_w.\texttt{value} = e_r.\texttt{value} \wedge e_r \overset{\texttt{hb}}{\not\longrightarrow} e_w$ (NONE if none exists)

          if e_w *is* NONE **then return** UNKNOWN

        Add $e_w \overset{\texttt{rf}}{\longrightarrow} e_r$ to rf and to hb

        repeat

          Apply co **inference** and rb **computation** (as in HEURISTIC-URF). *If adding an edge would introduce a cycle, return* UNKNOWN *instead of* INCONSISTENT.

        until *fixpoint of* hb *reached*

  foreach address $\in dom(\mathcal{W})$ **do** `// --- completing the co relation ---`

    foreach *distinct* $e_{w1}, e_{w2} \in \mathcal{W}(\texttt{address})$ **do**

      if $e_{w1} \overset{\texttt{hb}}{\not\longrightarrow} e_{w2}$ ***and*** $e_{w2} \overset{\texttt{hb}}{\not\longrightarrow} e_{w1}$ **then**

        if $e_{w1} \overset{\texttt{hb}_{\text{pre}}}{\longrightarrow} e_{w2}$ **then** add $e_{w1} \overset{\texttt{co}}{\longrightarrow} e_{w2}$ to hb

        if $e_{w2} \overset{\texttt{hb}_{\text{pre}}}{\longrightarrow} e_{w1}$ **then** add $e_{w2} \overset{\texttt{co}}{\longrightarrow} e_{w1}$ to hb

  Add edges $\texttt{rb} \leftarrow \texttt{rf}^{-1}$; co to hb. *If adding some edge* $e_1 \overset{\texttt{hb}}{\longrightarrow} e_2$ *would create a cycle (equivalently if* $e_2 \overset{\texttt{hb}}{\longrightarrow} e_1$ *already exists) return* UNKNOWN `// --- rb computation ---`

  $\widetilde{E} \leftarrow$ TOPOLOGICALSORT(hb)

  return $\widetilde{E}$

return UNKNOWN

Whenever adding an edge would result in a cycle, the heuristic conservatively returns Unknown, even though an alternative choice of `rf` or `co` might have yielded consistency. This is unavoidable: deciding execution feasibility under SC is NP-complete, and achieving completeness in polynomial time would imply P = NP.

In practice, Heuristic-INCR successfully constructs SC prefixes for the vast majority of feasible candidate read-cuts, thereby substantially reducing the number of invocations of ConstructWitnessComplete.

5.4 Final ConstructWitness Algorithm

By combining the heuristics with the complete algorithm, the final ConstructWitness algorithm is obtained. Its pseudocode is given in Algorithm 6.

Algorithm 6: ConstructWitness(E, e_r^{new})

Input: Sequence of operations E, new read operation e_r^{new}

Output: Sequentially consistent execution prefix $\widetilde{E}$ such that $\widetilde{E}$ is a reordering of E, or Inconsistent

```
1 U ← Heuristic-URF(E)
2 if U = Inconsistent then return Inconsistent
3 else (urf, hb) ← U
4 Ẽ ← Heuristic-INCR(E, e_r^new, (urf, hb))
5 if Ẽ ≠ Unknown then return Ẽ
6 return ConstructWitnessComplete(E)
```

6 Why Naively Adapting the Reads-From Exploration Fails

At first glance, sound and optimal stateless model checking (SMC) under view equivalence may appear straightforward. For example, one natural idea is to adapt techniques developed for reads-from equivalence. In particular, algorithms such as Optimal SMC for Reads-From Equivalence [5] explore new executions by changing the source write of a single read operation while holding the rest of the reads-from relation fixed. This strategy succeeds under reads-from equivalence because the reads-from relation uniquely characterizes each equivalence class.

A straightforward extension to view equivalence would attempt to explore new view equivalence classes by altering the values read by one read at a time, while keeping the others unchanged, achieved by fixing the reads-from relation for all but that read.

However, this seemingly minor restriction introduces a critical flaw. By preserving the reads-from edges of all other reads, the algorithm also preserves the induced happens-before relation. Since happens-before constraints determine which writes are visible to a read, the algorithm may fail to realize executions where a new combination of values is achievable only if *multiple reads simultaneously change their source writes*. In such cases, no single read-from mutation suffices.

Example 6.1. Consider the following program with three threads and two shared variables `x` and `y`:

p	q	r
`x := 2`	`y := 3;`	`a := x;`
	`y := 4;`	`b := y`
	`x := 2`	

Suppose we explore an execution where r_1 reads from q_3. Program order then enforces the following happens-before chain:

$$q_1 \xrightarrow{\texttt{po}} q_2 \xrightarrow{\texttt{po}} q_3 \xrightarrow{\texttt{rf}} r_1 \xrightarrow{\texttt{po}} r_2$$

As a result, r_2 happens-after q_2, which itself happens-after q_1. Therefore, if r_1 reads from q_3, then r_2 cannot read from q_1. Thus, if the algorithm attempts to force r_2 to read `3` while keeping r_1 fixed to reading from q_3, it finds no valid execution and incorrectly concludes that `3` is an impossible value for r_2.

Yet such an execution does exist. Consider:

$$p_1 q_1 r_1 r_2 q_2 q_3$$

Here, r_1 reads from p_1, scheduled before q_3. The happens-before relation no longer prevents q_2 from reading from q_1. These constraints are illustrated in Fig. 4.

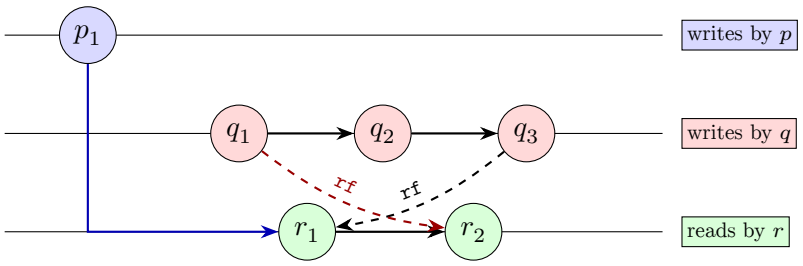

Figure 4: Two competing *tentative* reads-from edges (dashed). If r_1 happens-after q_3, the `hb` chain blocks the red edge; moving p_1 earlier unlocks it and lets q_2 read 3.

This example illustrates the core limitation of this strategy: by mutating the reads-from write of only a single read while keeping all others fixed, causality constraints are introduced that can mask feasible executions. As a result, this strategy is *unsound* under view equivalence, since it may miss valid view equivalence classes.

One might attempt to address this limitation by dynamically rescheduling or injecting additional operations to "unblock" hidden writes; for example, by scheduling p_1 just before r_1 so that r_1 reads from p_1 instead of q_3. This adjustment would remove the happens-before constraint that previously blocked r_2 from reading from q_1.

While this may succeed in simple examples, it does not resolve the fundamental issue. The entire exploration strategy is predicated on keeping the remaining reads-from edges fixed. This fixed context may encode deep causal dependencies that cannot be broken by local rescheduling.

For example, writes cannot always be arbitrarily rescheduled due to control-flow constraints, conditional execution, and data dependencies. For instance, the write p_1 might be able to move only if an earlier read of p on an unrelated variable `z` reads-from a particular write. Moving p_1 would then require altering the source of this read, which in turn may cascade to other parts of the execution. Thus, what begins as an attempt to change the value of a single read, can force consideration of unrelated reads and writes – potentially to entirely different variables – just to reach a viable execution. This highlights the inherent difficulty of constructing sequentially consistent executions under view equivalence: even a local change in one read's value may necessitate a global reconfiguration of the execution's causal structure.

This discussion highlights a fundamental distinction between reads-from and view equivalence. Under reads-from equivalence, exploration can proceed incrementally by mutating the source of a single read while keeping all others fixed. Under view equivalence, by contrast, sound and complete exploration may require mutating sources of multiple reads simultaneously. Consequently, any viable exploration algorithm must be able to dynamically adjust causality across the entire execution.

7 Implementation

7.1 Forks and Joins

Practical runtimes such as `pthread` permit dynamic thread creation and joining. We extend our model to handle `fork` and `join` operations as follows.

Let e_{fork} denote a spawn operation run by the parent thread, p^{new} the new thread, $e^{\mathsf{new}}_{\mathsf{first}}$ the first operation of the new thread, $e^{\mathsf{new}}_{\mathsf{last}}$ its last operation, and e_{join} a join operation. We add the constraints:

$$e_{\mathsf{fork}} \xrightarrow{\mathsf{po}} e^{\mathsf{new}}_{\mathsf{first}} \qquad \text{and} \qquad e^{\mathsf{new}}_{\mathsf{last}} \xrightarrow{\mathsf{po}} e_{\mathsf{join}}$$

These constraints are injected into `po` wherever it is constructed:

- in Heuristic-URF and Heuristic-INCR, when initialising `hb`, and
- in ConstructWitnessComplete, as inequalities $V_{e_{\mathsf{fork}}} < V_{e^{\mathsf{new}}_{\mathsf{first}}}$ and $V_{e^{\mathsf{new}}_{\mathsf{last}}} < V_{e_{\mathsf{join}}}$.

Equivalent encoding. For theoretical purposes, it is useful to observe that fork and join can be expressed entirely in terms of ordinary memory operations. In this encoding:

- e_{fork} is modelled as a write to a fresh location with a fresh value. A virtual read, prepended to $E_{p^{\mathsf{new}}}$, reads the fresh location. This virtual read is, thus, constrained to always `rf` from e_{fork}.
- A virtual write, appended to $E_{p^{\mathsf{new}}}$, writes to a second fresh location with a fresh value. e_{join} is modelled as a read that reads the fresh location. e_{join} is, thus, constrained to always `rf` from this virtual write.

This encoding enforces exactly the same causal constraints as the explicit $\xrightarrow{\mathsf{po}}$ edges:

- The child thread cannot perform any operations before the parent's fork, since the virtual read in p^{new} depends on the fork's virtual write.
- A join cannot occur until the corresponding thread has terminated, since the join, as a virtual read, depends on the final virtual write in p^{new}.

Thus, programs with dynamic thread creation and joining can be reduced to programs containing only ordinary reads and writes. This equivalence implies that our SC model requires no additional axioms, and all theorems proved for the base model extend unchanged to the setting with fork and join.

7.2 URF and INCR: Graph Representations

Both Heuristic-URF and Heuristic-INCR maintain explicit representations of the `rf` and `hb` relations.

Reads-from. The `rf` relation is represented as a dense array indexed by execution position:

$$\texttt{rf}[i] = \begin{cases} j & \text{if operation } i \text{ is a read and reads from operation } j, \\ -1 & \text{if operation } i \text{ is a write or a read that is currently unmatched} \end{cases}$$

This representation ensures constant-time lookup and efficient updates.

Happens-before. The `hb` relation is heavily queried for reachability, so we store it as its transitive closure. We experimentally evaluated edge lists, adjacency lists, and adjacency matrices. Because the transitive closure tends to be dense in practice, the adjacency-matrix representation is the most efficient.

Let $n = |E|$. We store `hb` as an $n \times n$ bit matrix, laid out in row-major order in machine-word blocks. Adding an edge (u, v) updates the closure using bitwise operations:

> Let P be the set of predecessors of u (rows x such that $x \to u$) and S the set of successors of v (columns y such that $v \to y$). Then for each $x \in P \cup \{u\}$ and $y \in S \cup \{v\}$, set $x \to y$.

In code, this amounts to:

$$\texttt{row}[x] \mid= \texttt{row}[x] \vee \texttt{row}[v]; \qquad \texttt{col}[y] \mid= \texttt{col}[y] \vee \texttt{col}[u]$$

implemented as word-wise OR operations which can be performed over contiguous memory if both the adjacency matrix and its transpose are maintained. These operations vectorise efficiently and exhibit good cache locality.

7.3 Nidhugg Integration

Nidhugg structures SMC algorithms around a stateful TraceBuilder object that is driven by the scheduler. This design is inverted relative to the classical presentation where the SMC algorithm invokes the scheduler. In Nidhugg, the scheduler is the main loop, while TraceBuilder supplies scheduling decisions and records memory operations.

During execution, the scheduler repeatedly invokes:

- TraceBuilder.schedule() to obtain the next thread to execute,
- TraceBuilder.load() and TraceBuilder.atomic_store() to record memory reads and writes,
- TraceBuilder.spawn() and TraceBuilder.join() to record thread creation and joining.

At the end of each execution, the scheduler calls TraceBuilder.reset(), and subsequently queries TraceBuilder.should_continue() to determine whether a new execution should be initiated.

To recover the familiar control flow where the SMC algorithm drives the scheduler without modifying Nidhugg, we implement the algorithm as a coroutine using `Boost::Context`:

- The algorithm yields whenever it wishes to invoke the scheduler, passing the execution prefix through a shared field in TraceBuilder.
- The scheduler consumes this prefix, executes it to completion, stores the resulting execution in TraceBuilder, and resumes the coroutine.

The coroutine maintains its own stack. Switching between the scheduler and the coroutine is implemented as a stack switch. This preserves fidelity to the algorithmic specification while integrating cleanly with Nidhugg's architecture.

7.4 Flyweight Pattern

Executions are sequences of operations. In practice, all executions are composed from a comparatively small set of distinct operations, combined in different orders. Because the EXECUTIONS container must hold many executions simultaneously, storing complete copies of these operation records for each execution would be prohibitively expensive. To eliminate this redundancy, we apply the *Flyweight* pattern: each distinct operation is stored once, and executions reference it via pointers to immutable operation instances (flyweights).

A *Flyweight Store* maintains a map from operation hashes to operation buffers. When a client requests an operation, it constructs a temporary buffer (e.g. on the caller's stack) and queries the *Flyweight Store*, which:

1. Computes a (collision-resistant) hash h of the buffer.
2. Looks up h in the map. If an entry is found, the existing pointer is returned.
3. Otherwise a fresh buffer is allocated on the heap, the temporary buffer is copied into it, ($h \mapsto$ buffer) is inserted into the map, and a pointer to the new buffer is returned.

Flyweights are immutable and managed by reference counting. The buffer associated with each operation additionally stores an integer reference count:

- On successful lookup or insertion, the store increments the reference count.
- On release, the store decrements it. If the count reaches 0, the buffer and its map entry are destroyed.

7.5 SMT Backend

The complete algorithm is implemented using Z3 via its official C++ API. Executions are encoded into quantifier-free integer difference logic (QF-IDL) formulas, and satisfiability queries are used to determine satisfiability.

7.6 An Optimized Membership Structure: The CONTAINSSET Data Structure

7.6.1 Motivation and High-Level Overview

In our implementation, the EXPLORED and TOBEEXPLORED sets are used exclusively for membership checks – there is no need to enumerate their contents or retrieve elements. This restricted usage enables a highly space-efficient representation. We introduce a simple yet powerful optimization, which we call the CONTAINSSET.

The CONTAINSSET is implemented as a conventional hash set with $O(1)$ average-case access time. However, rather than storing full elements, it stores only a fixed-size cryptographic hash of each item. This guarantees **constant space per element**, irrespective of its actual structure or size.

7.6.2 Cryptographic Hashing and Collision Resistance

To achieve this, we employ a cryptographically secure hash function such as SHA2 or BLAKE3. These functions offer *collision resistance*, a well-established cryptographic property: no efficient (i.e., probabilistic polynomial-time) algorithm is known to generate two distinct inputs with the same hash

output. More precisely, any probabilistic polynomial-time (PPT) algorithm can produce two distinct inputs that hash to the same value with only negligible probability in the size of the hash output – i.e., for every polynomial $p(n)$, there exists an N such that for all output lengths $n > N$, the success probability is less than $1/p(n)$. Thus, the likelihood of collision is essentially negligible in practice.

To date, no known collision has been found for any widely used cryptographically secure hash function such as SHA2. In the unlikely event that a hash function is broken, it can be substituted with another secure hash function. It is important to note that cryptographic hash functions differ fundamentally from the hash functions used in standard hash sets, which are not designed to completely eliminate collisions.

7.6.3 Random Oracle Modeling and Statistical Guarantees

Cryptographic hash functions are frequently modeled using the *Random Oracle Model* (ROM), an idealized framework introduced to analyze the security of cryptographic constructions. In this model, the hash function H is treated not as a deterministic algorithm, but as an abstract oracle – a black box – that answers each query with a random response, subject to the constraint of consistency.

Formally, the random oracle is a function $H : \{0,1\}^* \to \{0,1\}^n$, where $\{0,1\}^*$ denotes the set of all finite-length binary strings (the input domain), and $\{0,1\}^n$ is the fixed-length output space (e.g., $n = 256$). The oracle maintains an initially empty internal table (or mapping). On receiving a query $x \in \{0,1\}^*$:

1. If x is already in the table, the oracle returns $H(x)$ as previously assigned.
2. If x is new, the oracle samples $H(x)$ uniformly at random from $\{0,1\}^n$, stores the pair $(x, H(x))$, and returns $H(x)$.

This process guarantees that each input yields an independently random output, and repeated queries to the same input return the same value, preserving determinism across calls. Crucially, the mapping from inputs to outputs is *statistically independent* for distinct inputs, and behaves as if it were a truly random function selected uniformly from all functions mapping $0,1^*$ to $0,1^n$.

Although true random oracles cannot exist in practice – since no real-world function can simulate such an idealized behavior – they serve as a powerful abstraction. Many cryptographic protocols and security proofs are constructed and verified under this model. While real-world hash functions (like SHA2, SHA3, or BLAKE3) are deterministic algorithms, they are designed to *heuristically approximate* random oracles in practice.

Despite being a heuristic model, the Random Oracle Model has been remarkably successful in cryptography: schemes proven secure under Random Oracle Model and instantiated with secure cryptographic hash functions remain unbroken, even when subjected to extensive cryptanalytic effort. As such, the Random Oracle Model remains a widely accepted and pragmatic foundation for reasoning about hash-based constructions in both theory and practice.

7.6.4 Quantitative Analysis of Collision Probability

Let us consider storing Q distinct elements in a ContainsSet, where the hash function maps each element to one of $N = 2^n$ possible outputs (e.g., $n = 256$ bits). Since the elements are distinct, the hash values are modeled as Q independent and uniformly random samples from the output space.

The probability that at least one collision occurs among a set of independently hashed elements is captured by the well-known *Birthday Problem*. A standard analysis of this probability – such as the one presented in Appendix A.4 of *Introduction to Modern Cryptography* by Jonathan Katz and Yehuda Lindell [16] – yields the following bounds:

$$\frac{Q(Q-1)}{4N} \leq 1 - \exp\left(-\frac{Q(Q-1)}{2N}\right) \leq \textsc{Coll}(Q, N) \leq \frac{Q(Q-1)}{2N} \tag{4}$$

where $\textsc{Coll}(Q, N)$ denotes the probability that at least one collision occurs among the Q hash outputs drawn uniformly at random from a space of size N.

7.6.5 Practical Implications of Collision Bounds

For instance, if $Q = 10^9$ and we use a 256-bit hash function (so $N = 2^{256}$), such as SHA2-256, then:

$$\textsc{Coll}(10^9, 2^{256}) < \frac{10^9(10^9 - 1)}{2 \cdot 2^{256}} \leq 4.4 \cdot 10^{-60}$$

That is, the probability of even a single collision is astronomically small.

To appreciate the scale: the fastest supercomputers today achieve approximately 10^{18} floating-point operations per second (1 exaFLOP) on benchmarks such as LINPACK, which involve solving large dense systems of linear equations. Although cryptographic hash computations are substantially more complex than the arithmetic operations involved in such workloads, let us assume – generously – that such a machine could perform 10^{18} cryptographic hash evaluations per second.

Suppose that in each experiment, we generate and hash 10^9 new, randomly chosen elements. With a collision probability bounded by $4.4 \cdot 10^{-60}$ per such experiment, it would take on average

$$\frac{1}{4.4 \cdot 10^{-60}} \approx 2.3 \cdot 10^{59} \text{ executions}$$

to observe a single collision. Since each execution involves 10^9 hashes, the total number of hashes required is $2.3 \cdot 10^{68}$. At a rate of 10^{18} hashes per second, this would take:

$$\frac{2.3 \cdot 10^{68}}{10^{18}} = 2.3 \cdot 10^{50} \text{ seconds},$$

which corresponds to roughly $7.3 \cdot 10^{42}$ years – approximately $5.3 \cdot 10^{32}$ times the age of the universe, estimated at $13.8 \cdot 10^9$ years.

7.6.6 Choosing the Hash Output Size

We can derive a general expression for the minimum required hash output length n (in bits) to bound the collision probability by some ε:

$$\log_2\left(\frac{Q(Q-1)}{4\varepsilon}\right) \leq n \leq \log_2\left(\frac{Q(Q-1)}{2\varepsilon}\right) \tag{5}$$

Under the common approximation $Q - 1 \approx Q$, this simplifies to:

$$2\log_2 Q - \log_2 \varepsilon - 2 \leq n \leq 2\log_2 Q - \log_2 \varepsilon - 1 \tag{6}$$

Thus, the upper and lower bounds differ by only 1 bit, giving us precise control over collision guarantees for a given Q and ε.

7.6.7 Implementation Details

Our present implementation employs the xxHash algorithm, selected primarily for its remarkable computational efficiency. It should be stressed, however, that xxHash is a non-cryptographic hash function, and thus its design does not aim to prevent the deliberate construction of colliding inputs. Consequently, one cannot exclude the possibility that an efficient collision-finding algorithm may exist. Nevertheless, in the absence of adversarially chosen inputs, as is the case in our application, such concerns are largely theoretical, and xxHash provides an attractive balance between performance and reliability in practice.

In our current implementation, we selected the 64-bit variant of xxHash, denoted XXH64. Based on experimental evidence, we do not expect to store more than 10^8 elements in a single ContainsSet. Under this assumption, and modeling XXH64 as a random oracle, the probability of a collision when inserting 10^8 distinct elements is bounded by:

$$\textsc{Coll}(10^8, 2^{64}) < \frac{10^8(10^8 - 1)}{2 \cdot 2^{64}} < 3 \cdot 10^{-4}.$$

This bound indicates that collisions are exceedingly unlikely to occur in practice.

For applications in which stronger guarantees against collisions are required, XXH64 can be replaced with a hash function that produces a larger output (e.g., 128 or 256 bits) and/or a cryptographically secure alternative such as SHA2-256 or BLAKE3-256. Increasing the output length exponentially reduces the collision probability, while cryptographic hash functions additionally provide collision-resistance guarantees that remain secure even in the presence of adversarially chosen inputs. In such cases, the probability of collision can be reduced to a level so small that it lies far beyond the reach of any conceivable computational effort, and collisions may be regarded as practically impossible.

8 Evaluation

We evaluate the performance of ViewXplore on the ReadInc program, running on an Intel Core i7 (1st Gen) CPU at 1.60 GHz. We compare ViewXplore against the following state-of-the-art SMC algorithms: Optimal-DPOR [1], Optimal-DPOR-Observers [7], SMC-RVF [6, 25], Data-Centric DPOR [8, 9], and Optimal-SMC-RF [5].

	ReadInc [n=6]	ReadInc [n=7]
#(Interleavings)	7,484,400	681,080,400
#(`hb` equivalence classes)	518,400	25,401,600
Optimal-DPOR time	85.63 s	5480.30 s
#(Observers equivalence classes)	157,717	6,053,748
Optimal-DPOR-Observers time	29.25 s	1432.92 s
DC-DPOR reported #(Executions)	84,682	2,625,219
DC-DPOR time	30.31 s	1225.83 s
SMC-RVF reported #(Executions)	14,495	181,103
SMC-RVF time	8.86 s	130.38 s
#(`rf` equivalence classes)	16,807	262,144
Optimal-SMC-RF time	2.70 s	51.40 s
#(View Equivalence Classes)	4,683	47,294
ViewXplore	**0.66 s**	**8.61 s**

Table 2: Performance comparison on the ReadInc benchmark. ViewXplore achieves the fewest equivalence classes and the lowest runtime among all methods.

As shown in Table 2, ViewXplore achieves a substantial reduction in both the number of explored executions and the total exploration time. Compared to Optimal-DPOR, it reduces the number of explored executions by nearly two-three orders of magnitude. Even against the more refined Optimal-DPOR-Observers and Optimal-SMC-RF, ViewXplore consistently explores fewer equivalence classes and completes significantly faster. For instance, on ReadInc [n=7], ViewXplore is over **160× faster** than Optimal-DPOR and roughly **6× faster** than Optimal-SMC-RF.

9 MCSC: Model Checking for Sequential Consistency is NP-complete

We establish the computational complexity of model checking under the Sequential Consistency (SC) memory model. Specifically, we show that the decision problem of determining whether any SC execution violates a safety property – referred to as MCSC – is NP-complete. To the best of our knowledge, this result has not previously appeared in the literature.

We first formalize the decision problem.

Definition 9.1 (MCSC decision problem)**.** *Given a program to be executed concurrently by n threads, determine whether there exists an interleaving (under the Sequential Consistency model) in which at least one thread crashes. The answer is* Yes *if such an interleaving exists, and* No *otherwise.*

A crash *is defined as follows: for each thread, there exists a set of sequences of read values, called the* Crashing Set*, such that the thread crashes if and only if the sequence of values it reads belongs to its* Crashing Set*. The* Crashing Sets *are not known to the model checker a priori; they can only be discovered by executing runs in which a thread reads such a sequence.*

More formally, the decision problem asks whether there exists a sequentially consistent execution in which the sequence of values read by some thread lies in its Crashing Set*.*

9.1 NP-hardness

We show NP-hardness via a Karp reduction from the well-known SubsetSum problem.

Definition 9.2 (SubsetSum decision problem)**.** *Given a finite set $A = \{a_1, a_2, \dots, a_n\}$ of integers and a target value $S \in \mathbb{Z}$, determine whether there exists a non-empty subset $A' \subseteq A$ such that $\sum_{a \in A'} a = S$.*

Theorem 9.1 (NP-hardness of MCSC)**.** MCSC *is* NP*-hard.*

Proof. Let (A, S) be an instance of SubsetSum, where $A = \{a_1, a_2, \dots, a_n\}$.

We construct a corresponding instance of MCSC as follows.

We introduce n threads $p^1, \dots, p^n$ – one per element of A – and a single shared variable `x`, initially set to 0. The concurrent program is the following:

p^1	p^2		p^n
`b := x;`	`b := x;`		`b := x;`
`assert(b + a1 ≠ S)`	`assert(b + a2 ≠ S)`	...	`assert(b + an ≠ S)`
`x := b + a1`	`x := b + a2`		`x := b + an`

This defines the MCSC instance.

We now show that the SubsetSum instance is a Yes-instance if and only if the MCSC instance is a Yes-instance, i.e., if the previous program admits an interleaving where a thread crashes.

($\Rightarrow$) If the SubsetSum instance is Yes-instance, then the MCSC instance is a Yes-instance. Suppose there exists a subset $A' = \{a_{i_1}, \dots, a_{i_k}\} \subseteq A$ such that $\sum_{a \in A'} a = S$. Consider

the interleaving in which threads $p^{i_1}, \ldots, p^{i_k}$ execute in sequence, each performing its entire program (read, assert, and write) in order. That is, the interleaving schedule is:

$$p_1^{i_1} p_2^{i_1} p_1^{i_2} p_2^{i_2} \cdots p_1^{i_k} p_2^{i_k}$$

where each thread is scheduled twice (once for its read and once for its write). Note that assertion checks are thread-local operations. All other threads, if any, execute after this interleaving in some arbitrary order, which is out of our interest.

Since all threads in this interleaving execute their reads and writes in sequence, the shared variable `x` accumulates the sum of the corresponding a_i values in A'.

The final thread in the sequence p^{i_k} reads the value $b = \sum_{a_j \in A' \setminus \{a_{i_k}\}} a_j$ and performs the assertion check $b + a_{i_k} \neq S$, which fails, as

$$b + a_{i_k} = \left(\sum_{a_j \in A' \setminus \{a_{i_k}\}} a_j \right) + a_{i_k} = \sum_{a_j \in A'} a_j = S$$

Therefore, thread p^{i_k} crashes, and the MCSC instance is a YES-instance.

($\Leftarrow$) If the MCSC instance is a YES-instance, then the SUBSETSUM instance is a YES-instance. Assume the MCSC instance is a YES-instance, that is, there exists an interleaving in which some thread p^k crashes.

We define that thread p^j *reads-from* thread p^i if p^j's read reads-from p^i's write, i.e., if p^i's write is the most recent write to `x` preceding p^j's read in the interleaving[8].

We now define the *reads-from chain* of a thread p^i, denoted $\mathcal{C}(p^i)$ as follows: Recursively prepend the thread that p^i reads-from, continuing until reaching a thread that reads the initial value $x = 0$.

It follows that the value read by a thread p^i is the sum of the a_i values associated with the threads in the chain $\mathcal{C}(p^i)$:

$$b = \sum_{p^j \in \mathcal{C}(p^i)} a_j$$

When p^k executes its assertion check, it computes

$$b + a_k = \sum_{p^j \in \mathcal{C}(p^k)} a_j + a_k = \sum_{p^j \in \mathcal{C}(p^k) \cup \{p^k\}} a_j$$

Since the assertion fails, this value must equal S. Thus, the subset

$$A' = \left\{ a_j \mid p^j \in \mathcal{C}(p^k) \cup \{p^k\} \right\} \subseteq A$$

is a solution to the original SUBSETSUM instance. Hence, the SUBSETSUM instance is a YES-instance.

This completes the reduction. Note that the size of the constructed program is linear in the size

[8] Since each thread performs exactly one read and one write, this relation is well-defined.

of the input (A, S), and each thread performs only simple arithmetic and one read/write to shared memory. □

9.2 On Execution Time of Programs

It is worth noting that, in general, proving that SC model checking is NP-hard is not particularly informative unless some restriction is placed on the execution time of the programs. Indeed, there exist programs that require exponential time to execute even a single run. In such cases, model checking becomes infeasible for trivial reasons. Furthermore, non-terminating programs trivially render model checking undecidable.

Hence, to obtain a meaningful complexity result, we restrict our attention to programs that execute in time polynomial in the size of the input – that is, programs whose total execution time (summed over all threads) is polynomial in their size. Under this assumption, the reduction given above remains valid: the constructed program consists of a linear number of simple arithmetic and control-flow operations, and each thread executes only two memory operations. Thus, the reduction applies even under this time-bounded restriction.

9.3 MCSC for polynomial-time programs is in NP

Having established NP-hardness, we now show that MCSC lies in NP under the aforementioned polynomial-time execution restriction, and thus conclude that MCSC is NP-complete.

Theorem 9.2 (MCSC in NP)**.** MCSC *lies in* NP *under the polynomial-time execution restriction.*

Proof. A valid certificate for a YES-instance consists of an sequence of thread identifiers that corresponds to an interleaving that leads to a crash. Since the total number of operations is polynomial in the input size, the certificate itself is of polynomial size.

The NP verifier operates as follows. Given a candidate certificate (interleaving) and the MCSC instance, the verifier simulates the execution of the concurrent program under the specified interleaving. This can be performed efficiently, assuming access to an interpreter for the concurrent programs. If the certificate includes invalid thread identifiers or results in any error other than a crash, the verifier returns NO. If the simulation completes without any thread crashing, the verifier also returns NO. Only if a crash occurs during simulation does the verifier return YES.

Since the verifier runs in polynomial time and accepts exactly those certificates that correspond to crashing executions, the problem is in NP. □

Corollary 9.1 (NP-completeness of MCSC)**.** MCSC *for polynomial-time programs is* NP*-complete.*

10 Future Work

We identify several directions for future work:

- **Support for locks and Read-Modify-Write (RMW) operations.** Extending our algorithm to handle mutual exclusion primitives and atomic Read-Motify-Write (RMW) operations.
- **Parallelization of exploration and witness construction.** The exploration of independent interleavings can be distributed across multiple cores, potentially leading to significant reductions in total runtime.
- **Extension to weaker memory models.** Incorporating relaxed consistency semantics such as Total Store Order (TSO), Partial Store Order (PSO), and Release-Acquire (RA) would allow our approach to capture the behavior of modern multiprocessor architectures.

List of Definitions

2.1 Definition (Observer Equivalence) . . . 30
2.2 Definition (Reads-From Equivalence) . . . 30
2.3 Definition (Soundness of an SMC algorithm) . . . 32
2.4 Definition (Optimality of an SMC algorithm) . . . 32
3.1 Definition (View Equivalence) . . . 34
4.1 Definition (Read-Cut) . . . 42
4.2 Definition (p-projection of execution E) . . . 42
4.3 Definition (Read-Cut of Execution) . . . 42
4.4 Definition (Execution-Induced Read-Cut) . . . 42
4.5 Definition (Feasible Read-Cut) . . . 42
4.6 Definition (Witness) . . . 42
4.7 Definition (rc-restriction of execution E) . . . 42
4.8 Definition (ConstructWitness) . . . 44
5.1 Definition (Unique Reads-From (URF)) . . . 56
9.1 Definition (MCSC decision problem) . . . 73
9.2 Definition (SubsetSum decision problem) . . . 73

List of Theorems

2.1 Theorem (SC Acyclicity Theorem [19]) . . . 26
2.2 Theorem (Number of SC interleavings with k threads, n operations each) . . . 28
3.1 Theorem (Equivalence classes in the MSV program) . . . 35
3.1 Corollary (Number of equivalence classes in the MSV program) . . . 36
3.2 Theorem (Asymptotic growth of equivalence classes in the MSV program) . . . 36
4.1 Theorem (Optimality of ViewXplore) . . . 47
4.2 Theorem (Soundness of ViewXplore) . . . 48
5.1 Theorem (Necessary and Sufficient Conditions for ConstructWitnessComplete) . . . 54
9.1 Theorem (NP-hardness of MCSC) . . . 73
9.2 Theorem (MCSC in NP) . . . 75
9.1 Corollary (NP-completeness of MCSC) . . . 75

List of Algorithms

1 ViewXplore Exploration Algorithm . . . 45
2 EnumerateReadCutsWithPruning(E) . . . 52
3 ConstructWitnessComplete(E) . . . 56
4 Heuristic-URF(E) . . . 58
5 Heuristic-INCR($E, e_r^{\text{new}}, (\texttt{urf}, \texttt{hb}_0)$) . . . 62
6 ConstructWitness(E, e_r^{new}) . . . 63

THIS PAGE INTENTIONALLY LEFT BLANK.

References

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. "Optimal Dynamic Partial Order Reduction". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. ACM, Jan. 2014, pp. 373–384. DOI: 10.1145/2535838.2535845. URL: http://dx.doi.org/10.1145/2535838.2535845.

[2] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. "Comparing Source Sets and Persistent Sets for Partial Order Reduction". In: *Models, Algorithms, Logics and Tools*. Springer International Publishing, 2017, pp. 516–536. ISBN: 9783319631219. DOI: 10.1007/978-3-319-63121-9_26. URL: http://dx.doi.org/10.1007/978-3-319-63121-9_26.

[3] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. "Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction". In: *Journal of the ACM* 64.4 (Aug. 2017), pp. 1–49. ISSN: 1557-735X. DOI: 10.1145/3073408. URL: http://dx.doi.org/10.1145/3073408.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. "Parsimonious Optimal Dynamic Partial Order Reduction". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Proceedings, Part II*. Vol. 14682. LNCS. Springer, July 2024, pp. 19–43. DOI: 10.1007/978-3-031-65630-9_2. URL: https://doi.org/10.1007/978-3-031-65630-9%5C_2.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. "Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3360576. URL: http://dx.doi.org/10.1145/3360576.

[6] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. "Stateless Model Checking Under a Reads-Value-From Equivalence". In: *Computer Aided Verification*. LNCS. Springer International Publishing, 2021, pp. 341–366. ISBN: 9783030816858. DOI: 10.1007/978-3-030-81685-8_16. URL: http://dx.doi.org/10.1007/978-3-030-81685-8_16.

[7] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. "Optimal Dynamic Partial Order Reduction with Observers". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, 2018, pp. 229–248. ISBN: 9783319899633. DOI: 10.1007/978-3-319-89963-3_14. URL: http://dx.doi.org/10.1007/978-3-319-89963-3_14.

[8] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. "Data-centric dynamic partial order reduction". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158119. URL: https://doi.org/10.1145/3158119.

[9] *Data-centric dynamic partial order reduction (Implementation)*. https://github.com/ViToSVK/nidhugg/tree/datacentric. GitHub repository, Git commit 3eabd1c29cacc85a219d60583a201b4e49c6185c.

[10] Cormac Flanagan and Patrice Godefroid. "Dynamic Partial-order Reduction for Model Checking Software". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL05. ACM, Jan. 2005, pp. 110–121. DOI: 10.1145/1040305.1040315. URL: http://dx.doi.org/10.1145/1040305.1040315.

[11] Phillip B. Gibbons and Ephraim Korach. "Testing Shared Memories". In: *SIAM Journal on Computing* 26.4 (Aug. 1997), pp. 1208–1244. ISSN: 1095-7111. DOI: 10.1137/s0097539794279614. URL: http://dx.doi.org/10.1137/S0097539794279614.

[12] Patrice Godefroid. "Software Model Checking: The VeriSoft Approach". In: *Formal Methods in System Design* 26.2 (Mar. 2005), pp. 77–101. DOI: 10.1007/s10703-005-1489-x. URL: http://dx.doi.org/10.1007/s10703-005-1489-x.

[13] Jeff Huang. "Stateless Model Checking Concurrent Programs with Maximal Causality Reduction". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. ACM, June 2015, pp. 165–174. DOI: 10.1145/2737924.2737975. URL: http://dx.doi.org/10.1145/2737924.2737975.

[14] Thomas Jones. *Mars Pathfinder: Priority Inversion Problem*. Tech. rep. NASA Jet Propulsion Laboratory, 1997. URL: https://www.cse.chalmers.se/~risat/Report_MarsPathFinder.pdf.

[15] Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. "Awaiting for Godot: Stateless Model Checking that Avoids Executions where Nothing Happens". In: *22nd Formal Methods in Computer-Aided Design*. Ed. by Alberto Griggio and Neha Rungta. FMCAD 2022. Trento, Italy: IEEE, Oct. 2022, pp. 284–293. DOI: 10.34727/2022/ISBN.978-3-85448-053-2_35. URL: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2%5C_35.

[16] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. en. 3rd ed. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press.

[17] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. "Effective Stateless Model Checking for C/C++ Concurrency". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3158105. URL: http://dx.doi.org/10.1145/3158105.

[18] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. "Truly Stateless, Optimal Dynamic Partial Order Reduction". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–28. DOI: 10.1145/3498711. URL: https://doi.org/10.1145/3498711.

[19] Ori Lahav and Viktor Vafeiadis. "Explaining Relaxed Memory Models with Program Transformations". In: *FM 2016: Formal Methods*. Springer International Publishing, 2016, pp. 479–495. ISBN: 9783319489896. DOI: 10.1007/978-3-319-48989-6_29. URL: http://dx.doi.org/10.1007/978-3-319-48989-6_29.

[20] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C–28.9 (Sept. 1979), pp. 690–691. ISSN: 0018-9340. DOI: 10.1109/tc.1979.1675439. URL: http://dx.doi.org/10.1109/TC.1979.1675439.

[21] Nancy G. Leveson and Clark S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[22] National Aeronautics and Space Administration (NASA). *Northeast Blackout of 2003 - Safety Message 2008-03-01*. Safety Message (PDF) from NASA Safety Center. Mar. 2008. URL: https://sma.nasa.gov/docs/default-source/safety-messages/safetymessage-2008-03-01-northeastblackoutof2003.pdf.

[23] Robert Nieuwenhuis and Albert Oliveras. "DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic". In: *Computer Aided Verification*. LNCS. Berlin Heidelberg: Springer, 2005, pp. 321–334. ISBN: 9783540316862. DOI: 10.1007/11513988_33. URL: http://dx.doi.org/10.1007/11513988_33.

[24] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. "Maximal Causal Models for Sequentially Consistent Systems". In: *Runtime Verification*. Ed. by Shaz Qadeer and Serdar Tasiran. Vol. 7687. LNCS. Berlin Heidelberg: Springer, 2013, pp. 136–150. ISBN: 9783642356322. DOI: 10.1007/978-3-642-35632-2_16. URL: http://dx.doi.org/10.1007/978-3-642-35632-2_16.

[25] *Stateless Model Checking Under a Reads-Value-From Equivalence (Implementation)*. https://github.com/ViToSVK/nidhugg/tree/reads_value_from. GitHub repository, Git commit 3f20b4f169a1bf26d50f478d2681ba23a605be8b.