Algorithmic methods for the verification of consistency in distributed systems

Rachid Zennou

Université de Paris

En cotutelle avec Université Mohammed V de RABAT

École doctorale de Sciences Mathématiques de Paris Centre (386)

Institut de recherche en informatique fondamentale (IRIF) &

Centre d'Etudes Doctorales en Sciences des Technologies de l'Information et de l'Ingénieur (ST2I)

École nationale supérieure d'informatique et d'analyse des systèmes (ENSIAS)

Algorithmic Methods for the Verification of Consistency in Distributed Systems

Par : **Rachid ZENNOU**

Thèse de doctorat d'informatique

Dirigée par Pr. Ahmed Bouajjani

Et par Pr. Mohammed Erradi

Présentée et soutenue publiquement le 24 mai 2021

Devant un jury composé de :

Hanan EL BAKKALI	Professeure	ENSIAS, Université Mohammed V Rabat	Présidente
Ahmed BOUAJJANI	Professeur	Université de Paris	Directeur
Mohammed ERRADI	Professeur	ENSIAS, Université Mohammed V Rabat	Directeur
Karim BAINA	Professeur	ENSIAS, Université Mohammed V Rabat	Rapporteur
Krishnan Narayan KUMAR	Professeur	Chennai Mathematical Institute, Inde	Rapporteur
Parosh AZIZ ABDULLA	Professeur	Université d'Uppsala, Suède	Rapporteur
Cezara DRAGOI	Chargée de recherche	INRIA, ENS Paris	Examinatrice

الحمدلله

الذي بنعمته تتم الصالحات

I dedicate this thesis work to my family. A special feeling of gratitude to my beloved parents, to my dearest brothers and sisters, and to my wonderful Love, without them, I could never have succeeded.

ᑕᓄᑐᕐᕕᑕ ᕐ ᘐᘐᓄ ᐱ ᒪᒪᓄ ᐅᐄᓄ ᓄᕐᑕᒪᓄ ᐱ ᕐᘓᑕᒪᓄ ᐅᐄᓄ ᑕᓄᓯᘐᕐᘐᑕ ᕐᐃᐅ.
ᐱᓄᘎᐦᐃ ᓄᘎᐟ ᑕᑕᕐᑐᘓᕐ ᑲᕐᕉᓄᐃ . ᓄᐱ ᕐᘎᐦᐃ ᔪᓄᐱᐃ ᑐᘐᘐᕐ . ᑕᓄᑐᕐᕕᑕ
ᐅᐄᓄ ᕐ ᑕᓄᒪᐃᓄ ᑕᓄᒪᑐᑐᓄᐊᑕ ᐅᐄᓄ ᑕᑐᓄᑕᓄᑐ ᐃ ᓄᕐᑕ ᑐᐅᐃᐅᐅ .

ᐃᓄᐃ ᒪᕐᐃᓄ ᐳᑐᕐᕐᐃᐃ:

ᐅᑐᕐᐄᕐ ᑲᓄ ᘐᐄᓄ ᑲᓄ

ᐅᐃᓄ ᕐᑐᓄᐃ ᑕᓄᒪᕐᒪᑕ ᕐᘓᘐᘐᑐ ᕐ ᒪᐅᖃᐅᘓ ᐃ ᑕᕐᑐᒪᓄ

ᒪᕐ ᕉᓄᐃ ᘎᓄ ᔪᓄᕼᕐ

ᓄᐃᘐᐱᐅ ᓄᕉ ᕐᐳᐱᒪᒪᒪ ᔪᓄᐃ ᕐ ᑕᕉᑐᘓᑕ

ᐅᑐ ᐱᓄ ᕐᑕᙆᙆᐃ ᑐᘐᘐᕐ ᔪᓄᐃ ᐄᘐᓄᘐ ᓄᑐ ᐱ ᕐᑫᐳᒪ ᘓᕐᐃ

Acknowledgments

Undertaking this PhD has been a challenge and a truly life-changing experience for me, and it would not have been possible to be fulfilled without the support and guidance that I received from many people.

First, I would like to express my sincere gratitude to my esteemed supervisors **Pr. Mohammed Erradi** and **Pr. Ahmed Bouajjani** for the continuous support and motivation, for the patience, and immense knowledge. Your guidance helped me in all the time of research and writing of this thesis. Your expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to become a good researcher, to sharpen my thinking and brought my work to a higher level. I could not have imagined having better advisors for my Ph.D. Thank you so much.

I would like to express my profound gratitude to the jury members: **Dr. Aziz Abdulla Parosh, Dr. Cezara Dragoi, Dr. Hanan EL BAKKALI, Dr. Karim BAINA,** and **Dr. Krishnan Narayan Kumar**, for having accepted to report this work and be part of the committee. I sincerely thank you for your availability and your precious remarks.

My thanks go to **Dr. Constantin Enea** for his availability, close collaboration, fruitful discussions and his kind assistance. I am thankful to him for giving me a strong background to pursue my research in this field and for helping me to achieve my goal. I also thank **Dr. Mohamed Faouzi Atig** for collaboration, discussions and availability.

I would like to thank my colleagues and friends at ENSIAS and IRIF. Especially, Ranadeep Biswas, Sidi Mohamed Beillahi, German Andres Delbianco, Suha Orhun Mutluergil, and Youness El ABDY, for discussions, help and support during my thesis. I would also like to thank the administrative staff at ENSIAS, Mohammed V University, ED 386, IRIF, and University of Paris for their help during my thesis. Especially, Pr. **MAHMOUD NASSAR.**

My warmest thanks go to all my teachers at Ghalil Amajyal Primary School, Abi Dhar Al Ghifari Middle School Ait Aissa Oubrahim, Aoulouz High School, Faculty of sciences and techniques Errachidia, Faculty of sciences Rabat, ENSIAS and IRIF. I deeply thank you for your guidance, advices, courses and lectures. I will forever be thankful to you.

Most importantly, my lovely family - without whom, this would never have been possible. My parents, I have no words to acknowledge your sacrifices. Thank you for your love, prayers, good education, endless support, encouragement and all things you did for me. May Allah grant you the highest place in Jannah. My Dearest brothers and sisters, you offered me invaluable support, love, advice, encouragement and everything needed through my entire life. You are the reason my childhood has been special, teenage

memorable, grown-up years unforgettable and my whole life remarkable. Thank you for supporting me in all my decisions and choices. From the bottom of my heart, thank you, My Love, for your encouragement, support, for being part of my life and for everything you have done for me. Thank you my little family “Ait Ben Amer”. I am fortunate to have such a wonderful family who were always with me inspiring to go for further study. I LOVE YOU ALL.

I would like also to say thank you to my uncles, aunts, cousins, and all my big family.

Thank you all my friends for all the unforgettable moments we spent together and for your support during all these years. Especially, Hamid, Youness, Mohssine, Wael, Amine, Ismail, Med, Moha, Mhammad, Badr, Youssef, Omar...

I would like to thank the NETYS/METIS organizing committee; it has been a great pleasure to be part of this committee. Especially, Pr. Khadija Bakkouch, Pr. Yahya Benkaouz, and Pr. Abdellah Boulouz.

This work was supported in part by CNRST “PBER: Programme de Bourses d’Excellence de Recherche ”, Campus France “Eiffel Excellence Scholarship Program”, IRIF, ANR project “AdeCoDS”, European Research Council (ERC), Agence universitaire de la Francophonie “Programme-pilote de soutien à l'excellence scientifique”, and ENSIAS, without either of which this work would not have been possible.

I would like to express my acknowledgments here to every person who have always supported me and contributed to the fulfillment of this work, to everyone who believed in me, and to All those Whom I Love & All those Who Love Me.

May all those who have contributed directly or indirectly to the elaboration of this work find here the expression of my sincere gratitude.

Abstract

Nowadays, we are all end users of distributed systems. A distributed system is a collection of computers in order to improve performance by sharing of resources. Indeed, with the internet's massive explosion, these systems have become necessary. Unfortunately, due to the parallelism and communication latency over large networks, distributed systems may produce unexpected (inconsistent) behaviors if they are not correctly designed and implemented. For instance, a flight seat can be assigned to two users of a flight booking system at the same time.

This thesis addresses the problem of verifying that an implementation of a concurrent/distributed system provides to the clients the expected consistency guarantees (i.e., strong, weak or eventual consistency). In particular, we consider the problem of testing concurrent/distributed systems to determine if they are offering the consistency level expected by their users. For a given computation of a concurrent/distributed system, the test confirms the consistency or inconsistency of the system during that computation. We propose dynamic verification approaches with respect to some well-known consistency models, i.e., executing a large number of test programs and verifying them against a given consistency model. The main consistency criterion that we consider in this thesis is a fundamental model called sequential consistency. The verification problem of this model is known to be NP-hard. The reason is that in order to prove that a computation is conform to this consistency model, one need to find a total order on write operations that explains the execution. Therefore, one need to enumerate all the possible total orders, in the worst case. In the beginning, we are interested in verifying the conformance to consistency models that are checkable in polynomial time using saturation-based techniques. We consider causal consistency in its different variants. Then, we build on this work in order to propose an approach for verifying sequential consistency using a strong causal consistency variant. This approach is improved by proposing another weaker model based on more natural and simpler saturation rules. These approaches allow to avoid falling systematically in the worst case i.e., enumerating explicitly the exponential number of the possible total orders between the computation writes. These two approaches are generalized afterward to cover another consistency model that is a relaxation of the sequential consistency model called total store ordering. The problem of verifying this model is also known to be NP-hard. Indeed, the

proposed generalizations use suitable models for approximating the total store ordering model. We implement all these approaches and perform benchmark on real life application.

Keywords: Formal verification, Verification, Testing, Consistency, Concurrency, Sequential consistency, Total store ordering, Causal Consistency, Distributed systems.

Résumé

Aujourd'hui, nous sommes tous des utilisateurs des systèmes distribués. Un système distribué est un ensemble d'ordinateurs afin d'améliorer les performances par le partage des ressources. En effet, avec l'explosion massive d'Internet, ces systèmes sont devenus nécessaires. Malheureusement, en raison du parallélisme et de la latence de communication sur les grands réseaux, les systèmes distribués peuvent produire des comportements inattendus (incohérents) s'ils ne sont pas correctement conçus et implémentés. Par exemple, un siège dans un vol peut être attribué à deux utilisateurs d'un système de réservation de vol au même temps.

Cette thèse aborde le problème de vérifier qu'une implémentation d'un système concurrent / distribué offre à ces clients les garanties de consistance attendues (consistance forte, faible ou éventuelle). En particulier, nous considérons le problème du test des systèmes concurrents / distribués pour déterminer s'ils offrent le niveau de consistance attendu par leurs utilisateurs. Pour une exécution d'un système concurrent / distribué donnée, le test confirme la consistance ou l'inconsistance du système lors de cette exécution. Nous proposons des approches de vérification dynamique par rapport à certains modèles de consistance très connus, i.e., en exécutant un grand nombre de programmes de test et en les vérifiant par rapport à un modèle de consistance donné. Le principal critère de consistance que nous considérons dans cette thèse est un modèle fondamental appelé la consistance séquentielle. Le problème de vérification de ce modèle est connu pour être NP-difficile. La raison est que, pour prouver qu'une exécution est conforme à ce modèle de consistance, il faut trouver un ordre total sur les opérations d'écriture qui l'explique. Par conséquent, il faut énumérer tous les ordres totaux possibles, dans le pire des cas. Au début, nous nous intéressons à vérifier la conformité à des modèles de consistance vérifiables en temps polynomial à l'aide de techniques basées sur la saturation. Nous considérons le modèle de la consistance causale dans ses différentes variantes. Ensuite, nous nous appuyons sur ces travaux pour proposer une approche de vérification de la consistance séquentielle en se basant sur une variante plus forte de la consistance causale. Cette approche est améliorée par la suite en proposant un autre modèle faible basé sur des règles de saturation plus naturelles et plus simples. Ces approches permettent d'éviter de tomber systématiquement dans le pire des cas i.e., énumérer explicitement le nombre exponentiel des ordres totaux possibles entre les écritures de l'exécution.

Ces deux approches sont ensuite généralisées pour couvrir un autre modèle de consistance qui est une relaxation de la cohérence séquentielle appelée "Total Store Ordering" (TSO). Le problème de la vérification de ce modèle est également connu pour être NP-difficile. En effet, la généralisation proposée utilise des modèles convenables pour approximer le modèle TSO. Nous allons implémenter toutes ces approches et réaliser des benchmarks sur des applications réelles.

Mots clés: Vérification formelle, Vérification, Test, Consistance, Concurrence, Consistance séquentielle, TSO, Consistance causale, Systèmes distribués.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

CAP	-	**C**onsistency, **A**vailability and **P**artition tolerance
SC	-	**S**equential **C**onsistency
TSO	-	**T**otal **S**tore **O**rdering
CC	-	Weak **C**ausal **C**onsistency
CCv	-	**C**ausal **Conv**ergence
CM	-	**C**ausal **M**emory
CCM	-	**C**onvergent **C**ausal **M**emory
wCCM	-	**w**eak **C**onvergent **C**ausal **M**emory
wSC	-	**w**eak **S**equential **C**onsistency
wTSO	-	**w**eak **T**otal **S**tore **O**rdering
CO	-	**C**ausal **O**rder
CF	-	**Conf**lict Order
CFe	-	**e**xternal **Conf**lict Order
PO	-	**P**rogram **O**rder
PPO	-	**P**reserved **P**rogram **O**rder
PO-LOC	-	**PO** per same **L**ocation
PWW	-	**P**artial **W**rite-**W**rite order
wPWW	-	**w**eak **P**artial **W**rite-**W**rite order
WR	-	**W**rite-**R**ead order
WRe	-	**e**xternal **W**rite-**R**ead relation
RW	-	**R**ead-**W**rite order
WW	-	**W**rite-**W**rite order
HB	-	**H**appen-**B**efore

wHB - **w**eak **H**appen-**B**efore
LHB - **L**ocal **H**appen-**B**efore
ST - **S**tore **O**rder
wST - **w**eak **S**tore **O**rder
NP - **N**ondeterministic **P**olynomial time
ADT - **A**bstract **D**ata **T**ype
ENUM - **ENUM**eration
SC-Ker - **S**equential **C**onsistency **Ker**nel

LIST OF PUBLICATIONS

This thesis presents the research work carried out during the Ph.D period. The outcomes of this work have been published in international conferences and journals.

- Zennou, R., Biswas, R., Bouajjani, A. et al. Checking causal consistency of distributed databases. Computing (2021). https://doi.org/10.1007/s00607-021-00911-3 **[Extended version of NETYS 2019 paper]**.
- Zennou R., Atig M.F., Biswas R., Bouajjani A., Enea C., Erradi M. (2020) Boosting Sequential Consistency Checking Using Saturation. In: Hung D.V., Sokolsky O. (eds) Automated Technology for Verification and Analysis. ATVA 2020. Lecture Notes in Computer Science, vol 12302. Springer, Cham. https://doi.org/10.1007/978-3-030-59152-6_20.
- Zennou R., Bouajjani A., Enea C., Erradi M. (2019) Gradual Consistency Checking. In: Dillig I., Tasiran S. (eds) Computer Aided Verification. CAV 2019. Lecture Notes in Computer Science, vol 11562. Springer, Cham: https://link.springer.com/chapter/10.1007/978-3-030-25543-5_16.
- Zennou R., Biswas R., Bouajjani A., Enea C., Erradi M. (2019) Checking Causal Consistency of Distributed Databases. In: Atig M., Schwarzmann A. (eds) Networked Systems. NETYS 2019. Lecture Notes in Computer Science, vol 11704. Springer, Cham: https://link.springer.com/chapter/10.1007/978-3-030-31277-0_3. **[Best student paper award]**.

CHAPTER 1

INTRODUCTION

A concurrent program defines operations that may be executed at the same time. This kind of programs are present at various levels of modern computer systems varying from distributed softwares to basic applications running on multicore systems, multi-tasking operating systems and multi-threaded programs. The design and implementation of concurrent systems is challenging and an error prone process because of the complexity of their behaviors resulting from the concurrency. Therefore, it is important to develop formal approaches to automatically check their correctness with respect to (w.r.t) some specifications. Several formal methods based technologies are proposed to verify these systems, for instance:

- Model checking [51]: is a method for checking whether a program executions meet given properties (specifications). The idea is to explore in an exhaustive and automatic way all models abstracting this program to decide if it satisfies the given specifications or not. One of the challenges for this approach is to handle the exponential number of possible states, commonly known as the "State Explosion Problem" [24].

- Deductive programs verification: also called programs proving, expresses the program correctness as a set of mathematical statements, known as verification conditions, based on the specifications that the program should meet. The SMT Solvers [15, 27] or interactive theorem provers [52, 16] are then used to discharge them.

- Static analysis: is performed on an abstraction of the given program in order to determine if it satisfies some particular properties. This method consists on analyzing the static program representation without executing it. Abstract interpretation introduced by P. Cousot and R. Cousot in [25, 26] is one of the widely used frameworks in static analysis.
- Dynamic analysis: consists on executing the program with test data and analyzing the product behaviors to assure that it fully satisfies all the expected specifications. Our thesis fits into the significant research effort for proposing efficient dynamic analysis algorithms for testing memory models conformance.

Motivation

The evolution of our modern society, based on the dramatic development of information and communication technologies, is closely linked to our growing need for automated services that have become crucial in all areas of our lives (communication, commerce, finance, transport, health, leisure, energy, etc.). With the emergence of the Internet of Things and Cloud Computing, there will be more and more connecting objects of all kinds, communicating and interacting through large-scale networks, having access to virtually unlimited computing power and memory resources. The deployment of these highly distributed systems and the control of their complexity give rise to enormous scientific and technological challenges.

The quest for performance pushes designers and developers of computer systems to use different kinds of optimization, and in particular to more and more parallelism and distribution, with a parsimonious use of synchronization. The general idea is to try to increase the throughput of the system, to make the data available and quickly accessible for clients, and to avoid the expectations due to blocking actions. This actually happens at all levels of computer systems, from the lowest level of multi-core hardware architectures to the highest level of distributed applications running on network infrastructures, including geo-replicated distributed databases.

These optimizations and the distributed nature of the calculations tend to reorder the actions performed by each of the system components. This may be due

to the fact that the more expensive actions (in terms of needed memory and/or computing power) are postponed, or performed in parallel, in order to allow the faster (or urgent) actions to be performed first. It can also be due to communication latency and the fact that messages can follow different paths across large networks. A system can then produce behaviors towards client programs that are not possible when all system actions are executed instantly and atomically, and are immediately visible to all processes in the system. This kind of behaviors correspond to the consistency models known as "strong consistency" models. In fact, strong consistency is generally difficult to ensure in an acceptable way from a performance point of view. Therefore, the majority of systems used in practice (both modern micro-processors, as well as platforms for cloud computing) implement weak consistency models. The relaxation of the consistency guarantees that a system provides to its clients (programs) may affect their correction. For instance, if a system implementing a distributed database is not highly consistent, this implies that the information on different replicas may be different at some point since updates are not immediately visible everywhere in the system.

This can affect the correction of applications using this system from both a safety and security perspectives. For example, two transactions on an account could withdraw twice the same amount available before they are executed if they are done in parallel and the updates are not immediately visible. Furthermore, if the policies for accessing information in a distributed database are updated in a weak manner i.e., these policies may differ from one site to another at a given moment, the information that are supposed to be protected can be leaked.

Then, one of the important problems is to ensure the correction of programs (clients) that will be executed on infrastructures that implement a weak consistency model. Indeed, concurrent and distributed systems are hard to design and program properly. This is due to the large number and complexity of interactions between their components. This difficulty become greater when these systems have to be executed according to a weak consistency model that allows even more behaviors that are complex, not intuitive, and extremely difficult to understand. Therefore, it is important to have methods and tools for automatically verifying the concurrent programs on weak consistency models.

A second compulsory problem is to verify whether a system that is supposed to provide a service according to a given consistency model is correctly implemented.

Thus, it is important to check that the consistency guarantees are well ensured (for clients) by the implementation. This is a crucial problem, especially with regard to object libraries and geo-replicated/distributed data structures, that are the building blocks for building modern infrastructures for cloud computing.

Contributions

The aim of this thesis is to study the two verification problems mentioned above and provide general and efficient solutions to solve them. The proposed solutions are generically applicable to a large spectrum of consistency models, especially those adopted for reasoning on distributed systems with replication. We briefly summarize the contributions of our thesis:

- First, we considered the problem of verifying that a computation is conform to a weak consistency model. We proposed an approach for verifying causal consistency models using a polynomial reduction of this problem to solving Datalog queries. Furthermore, we implemented our approach in an efficient tool for testing distributed systems.

- Then, we addressed the problem of verifying strong consistency models. We considered the fundamental model known as Sequential Consistency (SC, for short) and proposed a gradual approach for checking the conformance of a computation to this model. This approach is based on a strengthening of all known causal consistency models that still polynomially checkable. Next, we improved this approach by proposing a more natural and efficient upper-approximation of SC which is checkable in polynomial time as well.

- Finally, we considered the problem of verifying the Total Store ordering model (TSO, for short) which is a weakening of SC. In fact, we generalized the SC approaches to cover TSO model by proposing suitable models for approximating it. The suggested generalizations of the approaches mentioned above are not trivial. In particular, because of the fact that these two models (Sequential Consistency and Total Store ordering) consider different kinds of relations, the latter relaxed some relations considered in the former.

Related work

The problem of checking whether a history is SC has been proved to be NP-hard by Gibbons and Korach [41]. Two recent works tackle this verification problem [18, 7] and prove the interesting result that when the number of threads is fixed, the problem of checking the conformance of a single computation to SC is polynomial time in the size of the computation. These papers introduce algorithms for SC checking based on a search for interleavings corresponding to valid SC executions. However, their works are limited to the case of a fixed number of threads while our work consider the general problem. Biswas and Enea in [18] consider also the problem of checking some other consistency models in the transactional systems context and prove some complexity results about these models.

The fact that checking whether a history satisfies TSO is also NP-hard has been proved by Furbach et al. [38]. The problem of verifying that a finite-state shared-memory implementations (over a bounded number of threads, variables, and values) has been shown to be undecidable by Alur et al. [13].

Several static techniques have been developed to prove that a shared-memory implementation (or cache coherence protocol) satisfies SC [8, 13, 23, 28, 29, 32, 37, 40, 44, 56, 58], however only few have addressed dynamic techniques such as testing and runtime verification (which scale to more realistic implementations).

There are several works that addressed the testing problem for related criteria, e.g., Linearizability. While SC requires that the operations in a history be explained by a linearization that is consistent with the program order, linearizability is requiring that such a linearization be also consistent with the real-time order between operations (linearizability is stronger than SC). The works in [63, 49] describe monitors for checking linearizability that construct linearizations of a given history incrementally, in an online fashion. This incremental construction cannot be adapted to SC since it strongly relies on the specificities of linearizability. Line-Up [21] performs systematic concurrency testing via schedule enumeration, and offline linearizability checking via linearization enumeration. The works in [36, 35] show that checking linearizability for some particular class of ADTs is polynomial time.

Jepsen [1] is a framework for distributed systems verification used to check

different consistency models from eventual consistency to linearizability by using random clients. However, it focuses only on specific types of violations for a given consistency memory model.

Emmi and Enea [34] consider the problem of checking weak consistency criteria, but their approach focuses on specific relaxations in those criteria, falling back to an explicit enumeration of linearizations in the context of a criterion like SC or TSO. Bouajjani et al. [19] consider the problem of checking causal consistency. They formalize the different variations of causal consistency we consider in this thesis and show that the problem of checking whether a history satisfies one of these variations is polynomial time.

The idea of using weaker approximations of a memory consistency model (TSO) in order to detect violations has been used, e.g., in [59]. In that paper the authors use a form of saturation that corresponds to a variant of causal consistency (similar to convergence consistency [20]). However, their method is not complete. Our work generalizes this idea of saturation in the framework of gradual consistency checking introduced in the first part of chapter 4 (Section 4.1) where SC is approximated using several variants of causal consistency (including a new one called CCM). Then, we improved this idea in the second part of chapter 4 (Section 4.2) using a stronger (weak) consistency model (called wSC and stronger than CCM). We generalized these approaches to cover TSO as well in chapter 5.

The McVerSi framework [33] addresses test generation problem i.e., finding clients that increase the probability of uncovering bugs in shared memory implementations. Their methodology for checking SC lies within the context of white-box testing, i.e., the user is required to annotate the shared memory implementation with events that define the store order in an execution. In the approach we follow, the implementation is treated as a black-box requiring less user intervention. The Jepsen framework [1] also addresses the problem of finding clients by using randomization (introducing faults randomly). Since the efficiency of this approach has been shown in a recent work [53], we follow it to generate the used executions in our experiments using random clients.

Organization of the thesis

In addition to the two introductory chapters, this thesis is divided into three chapters in which we present our contributions in details.

In Chapter 2, we introduce the preliminaries. We recall basic definitions about binary relations and we present the used system model. Then, we recall the definitions of the consistency models considered in this thesis.

In chapter 3 of this thesis, we consider the problem of verifying weak consistency models. Indeed, we propose an approach and a tool for verifying causal consistency in its different variants. Then, we show that our approach is efficient and scalable by using it to verify real life distributed databases.

In the first part of chapter 4, we present an approach for checking Sequential Consistency gradually. The idea is to start by checking a weak consistency model that is stronger than all known causal consistency models. Then, if this model is not violated, the partial store order, that is computed using that model, is completed by enumeration to a total order. We show that our approach is more efficient compared to a standard enumeration using a SAT solver. The second part of this chapter presents a more efficient approximation for Sequential Consistency. This approximation is based on a stronger model compared to the one considered in the first part. Therefore, it allows, in addition of capturing more SC violations early, computing a large subset of the store order we need to find in order to prove Sequential Consistency conformance, if it exists. The experiment results show that the second approach outperforms the first one.

The chapter 5 introduces a generalization of the approaches proposed for Sequential Consistency to cover the case of Total Store Ordering model. We focus on finding suitable approximations for this model. Similarly to SC approaches, we show that these approximations perform good experimental results.

Finally, conclusions and perspectives are drown in chapter 6.

CHAPTER 2
FUNDAMENTALS

The first chapter is dedicated to the definitions of notations used through this document. First we present some preliminaries. Then, we give the system model used in this work. Afterwards, we define the notion of history and some related notions. Finally, we recall the consistency models we have studied. The rest of concepts that are used only locally in each chapter will be presented whenever they are needed.

2.1 Preliminaries

We now introduce the basic notions that we used in this thesis. First, we define some notions on binary relations.

2.1.1 Binary Relations

Given a set O and a binary relation $\mathcal{R} \subseteq O \times O$, we use the notation $(o_1,o_2) \in \mathcal{R}$ to denote the fact that o_1 and o_2 are related by $\mathcal{R}$. If $\mathcal{R}$ is an order, it denotes the fact that o_1 precedes o_2 in this order.
A binary relation $\mathcal{R} \subseteq O \times O$ is a strict order if it is

1. Irreflexive: for any $o \in O$, $(o, o) \in \mathcal{R}$ does not hold,
2. Asymmetric: for any o_1, $o_2 \in O$, if $(o_1, o_2) \in \mathcal{R}$, then $(o_2, o_1) \in \mathcal{R}$ does not hold, and

3. Transitive: for any o_1, $o_2 \in O$, if $(o_1, o_2) \in \mathcal{R}$ and $(o_2, o_3) \in \mathcal{R}$, then $(o_1, o_3) \in \mathcal{R}$.

A strict order $\mathcal{R}$ is total if, for any o_1, $o_2 \in O$, we have either $(o_1, o_2) \in \mathcal{R}$, $(o_2, o_1) \in \mathcal{R}$, or $o_1{=}o_2$.

For a binary relation $\mathcal{R} \subseteq O \times O$ over a given set O, we use $\mathcal{R}^+$ (resp. $\mathcal{R}^*$) to denote the transitive (resp. reflexive transitive) closure of $\mathcal{R}$. We use $\mathcal{R}^{-1}$ to denote the inverse relation of $\mathcal{R}$ (i.e., $(a, b) \in \mathcal{R}^{-1}$ iff $(b, a) \in \mathcal{R}$). We say that $\mathcal{R}$ is a partial order if it is irreflexive (i.e., $(a, a) \notin \mathcal{R}$ for all $a \in A$). We say that $\mathcal{R}$ is total if, for every $a, b \in A$, we have either $(a, b) \in \mathcal{R}$ or $(b, a) \in \mathcal{R}$. For two binary relations $\mathcal{R}_1$ and $\mathcal{R}_2$, we use $\mathcal{R}_1 \circ \mathcal{R}_2$ (resp. $\mathcal{R}_1 \cup \mathcal{R}_2$) to denote the composition (resp. union) of $\mathcal{R}_1$ and $\mathcal{R}_2$, i.e., $(a, b) \in \mathcal{R}_1 \circ \mathcal{R}_2$ iff there is $c \in A$ such that $(a, c) \in \mathcal{R}_1$ and $(c, b) \in \mathcal{R}_2$ (resp. $(a, b) \in \mathcal{R}_1 \cup \mathcal{R}_2$ iff $(a, b) \in \mathcal{R}_1$ or $(a, b) \in \mathcal{R}_2$).

Let O' be a subset of O. Then $\mathcal{R}_{|O'}$ is the relation $\mathcal{R}$ projected on the set O', that is $\{(o_1, o_2) \in \mathcal{R} \mid o_1, o_2 \in O'\}$. The set $O' \subseteq O$ is said to be *downward-closed* w.r.t a relation $\mathcal{R}$ if $\forall o_1, o_2$, if $o_2 \in O'$ and $(o_1, o_2) \in \mathcal{R}$ then $o_1 \in O'$ as well. A relation $\mathcal{R} \subseteq O \times O$ is a *strict partial order* if it is transitive and irreflexive. Given a strict partial order $\mathcal{R}$ over O, a *poset* is a pair $(O, \mathcal{R})$. Notice here that we consider the strict version of posets (not the ones where the underlying partial order is *weak*, *i.e.* reflexive, transitive and antisymmetric). Given a set Σ, a poset $(O, \mathcal{R})$, and a labeling function $\ell : O \to \Sigma$, the Σ *labeled poset* ρ is a tuple $(O, \mathcal{R}, \ell)$.

We say that ρ' is a *prefix* of ρ if there exists a downward closed set $A \subseteq O$ w.r.t. relation $\mathcal{R}$ such that $\rho' = (A, \mathcal{R}, \ell)$. If the relation $\mathcal{R}$ is a strict total order, we say that a (resp., labeled) *sequential poset* (sequence for short) is a (resp., labeled) poset. The concatenation of two sequential posets e and e' is denote by $e.e'$.

Consider a set of methods $\mathbb{M}$ from a domain $\mathbb{D}$. For $m \in \mathbb{M}$ and $arg, rv \in \mathbb{D}$, and $o \in O$, $\ell(o) = (m, arg, rv)$ means that operation o is an invocation of m with input arg which returns rv. The label $\ell(o)$ is sometimes denoted $m(\mathrm{arg}, rv)$. Let $\rho = (O, \mathcal{R}, \ell)$ be a $\mathbb{M} \times \mathbb{D} \times \mathbb{D}$ labeled poset and $O' \subseteq O$. We denote by $\rho\{O'\}$ the labeled poset where we only keep the return values of the operations in O'. Formally, $\rho\{O'\}$ is the $(\mathbb{M} \times \mathbb{D}) \cup (\mathbb{M} \times \mathbb{D} \times \mathbb{D})$ labeled poset $(O, \mathcal{R}, \ell')$ where for all $o \in O'$, $\ell'(o) = \ell(o)$, and for all $o \in O \setminus O'$, if $\ell(o) = (m, arg, rv)$, then

$\ell'(o) = (m, arg)$. Now, we introduce a relation on labeled posets, denoted $\preceq$. Let $\rho = (O, \mathcal{R}, \ell)$ and $\rho' = (O, \mathcal{R}', \ell')$ be two posets labeled by $(\mathbb{M} \times \mathbb{D}) \cup (\mathbb{M} \times \mathbb{D} \times \mathbb{D})$ (the return values of some operations in O might not be specified). The notation $\rho' \preceq \rho$ means that ρ' has less order and label constraints on the set O. Formally, $\rho' \preceq \rho$ if $\mathcal{R}' \subseteq \mathcal{R}$ and for all operation $o \in O$, and for all $m \in \mathbb{M}$, $arg, rv \in \mathbb{D}$, $\ell(o) = \ell'(o)$, or $\ell(o) = (m, arg, rv)$ implies $\ell'(o) = (m, arg)$.

2.1.2 System model

We consider multi-threaded programs over a set of shared variables $\mathsf{Var} = \{x, y, \ldots\}$. We assume that the set of (visible) operations issued by the threads of the program are read and write operations. Assuming an unspecified set of values Val and a set of operation identifiers OId, we let

$$\mathsf{Op} = \{\mathsf{read}_i(x, v), \mathsf{write}_i(x, v) : i \in \mathsf{OId}, x \in \mathsf{Var}, v \in \mathsf{Val}\}$$

be the set of operations reading a value v or writing a value v to a variable x. We omit operation identifiers when it is clear from the context. The set of read operations in a set of operations O is denoted by $\mathbb{R}(O)$. The set of write, operations in a set of operations O is denoted by $\mathbb{W}(O)$. The variable accessed by an operation o is denoted by $\mathsf{var}(o)$. Given a binary relation $\mathcal{R}$ on operations, let $\mathcal{R}_{\mathbb{WW}}$, $\mathcal{R}_{\mathbb{WR}}$, respectively, denote the projection of R on pairs of writes, pairs of writes and reads respectively, on the same variable.

2.1.3 Histories

We consider an abstract notion of an execution called *history* which includes a set of write or read operations ordered according to a (partial) *program order* po which order operations issued by the same thread. Most often, po is a union of sequences, each sequence containing all the operations issued by some thread. Then, we assume that the history includes a *write-read* relation wr which identifies the write operation writing the value returned by each read in the execution. Formally,

Definition 1 *A* history $\langle O, \mathsf{po}, \mathsf{wr} \rangle$ *is a set of operations* O *along with a strict*

partial program order po *and a* write-read *relation* $\mathsf{wr} \subseteq \mathbb{W}(O) \times \mathbb{R}(O)$, *such that the inverse of* wr *is a total function and if* $(\mathit{write}(x, v), \mathit{read}(x', v')) \in \mathsf{wr}$, *then* $x = x'$ *and* $v = v'$.

We assume that every *history* includes a write operation writing the initial value for each variable x. These write operations precede all other operations in po. Mentioning that these initial write operations is omitted when it is clear from the context. We notice that all considered histories are differentiated.

Differentiated histories. A history $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ is differentiated if each value is written at most once, i.e., for all write operations $\mathsf{write}(x, v)$ and $\mathsf{write}(x, v')$, $v \neq v'$. This is not a restriction since shared-memory implementations are data-independent [64] in practice.

Data Independence. An implementation is data-independent if its behavior does not depend on the concrete values read or written in the programs, and therefore any potential buggy behavior can be exposed by executions where each value is written at most once. We consider implementations that are data-independent which is a natural assumption that corresponds to a wide range of existing implementations. Thus, under this assumption, it is good enough to consider differentiated histories [19].

We use h, h_1, h_2, ... to range over histories. Since the writes on a variable are unique in the differentiated histories, the *write-read* relation can be easily extracted by only looking to the value fetched by each read operation.

Example 1 *The figure 2.1 presents a differentiated history in which the thread* t_1 *writes the value 1 on the variable* x, $\mathit{write}(x, 1)$, *then reads the value 1 from* y. *Similarly, the thread* t_2 *writes the value 1 on the variable* y, $\mathit{write}(y, 1)$, *then reads the value 1 from* x. *The* $\mathit{write}(x, 1)$, *resp.* $\mathit{write}(y, 1)$, *precedes* $\mathit{read}(y, 1)$, *resp.* $\mathit{read}(x, 1)$, *in the program order* po. *The* $\mathit{write}(x, 1)$, *resp.* $\mathit{write}(y, 1)$ *and* $\mathit{read}(x, 1)$, *resp.,* $\mathit{read}(y, 1)$ *are related by the write-read relation* wr, *i.e., the read* $\mathit{read}(x, 1)$, *resp.* $\mathit{read}(y, 1)$ *returns the value written by* $\mathit{write}(x, 1)$, *resp.* $\mathit{write}(y, 1)$.

t_1:	t_2:
write$(x, 1)$	write$(y, 1)$
read$(y, 1)$	read$(x, 1)$

Figure 2.1 – Differentiated history. Operations of the same thread are aligned vertically.

2.2 Consistency models

A memory consistency model defines a set of rules that determines how the system deals with operations from multiple processes (threads). In other words, it defines the possible return values of read operations. The consistency model can also be defined as a contract between programmers and system which defines the consistency guarantees that the programmers expect from the system. There exists several consistency models. In the following, we present the models we studied in this thesis from the strong consistency models to the relaxed consistency models. We first of all define some needed notions in the next sections.

Specification. The consistency of an object is defined w.r.t. a specification, determining the correct behaviors of that object in a sequential setting. In this work, we consider the read/write memory for which the specification S_{RW} is inductively defined as the smallest set of sequences closed under the following rules (x ∈ Var and v ∈ Val):

1. $\varepsilon \in S_{RW}$,
2. if $\rho \in S_{RW}$, then $\rho.\mathsf{write}(x, v) \in S_{RW}$,
3. if $\rho \in S_{RW}$ includes no write on variable x, then $\rho.\mathsf{read}(x, 0) \in S_{RW}$,
4. if $\rho \in S_{RW}$ and $\mathsf{write}(x, v)$ is the last write on variable x in ρ, then $\rho.\mathsf{read}(x, v) \in S_{RW}$.

Consistency models comparison.

- A consistency model M_1 is **stronger** than a consistency model M_2 if each possible computation under M_1 is also allowed under M_2. We say also that M_2 is **weaker** than M_1.
- Two consistency models M_1 and M_2 are **incomparable** if:
  - Some computation C_1 is valid (possible) on M_1 and not valid on M_2. And,

- Some computation C_2 is valid (possible) on M_2 and not valid on M_1.

Memory operations ordering. There exists four kinds of possible orders for memory operations:

- W-R: write operation must finish before the succeeding read operation.
- W-W: write operation must finish before the succeeding write operation.
- R-W: read operation must finish before the succeeding write operation.
- R-R: read operation must finish before the succeeding read operation.

2.2.1 From Strong to weak consistency

In order to ensure high availability and fault tolerance, distributed systems store data in more than one location i.e., replication. Then, the updates are sent to one replica (the nearest one, for instance) which forwards them to the other replicas. The advantage of the replication is that if a replica crush, the others can continue providing service. However, the question is how to keep replicas up to date i.e., consistent?

The problem is that considering network failures, distance, etc., we may have inconsistent replicas. To illustrate this problem, lets consider 3 users (Alice, Bob and David) in a messaging application group (Whatsapp for instance), which stores data in a distributed database. As these users are in the same group, any message sent by one of them will be forwarded to all the others.

Imagine now that David send a message to say that he is in Marrakech this week, then Alice answers "So good for you", afterwards, Bob says that he loses his phone. The ideal for users is to observe all messages in the same order so they can all understand what happened (Figure 2.2). This behavior can, theoretically, be guaranteed by strong consistency which guarantees that all replicas have the same state all time i.e., updates should be seen in the same order by all replicas.

Figure 2.2 – Strong consistent messages in a messaging application.

However, this is actually impossible in existence of failures. So, let's see what can happen in reality. Imagine now that bob receives the Alice message before the David message (Figure 2.3).

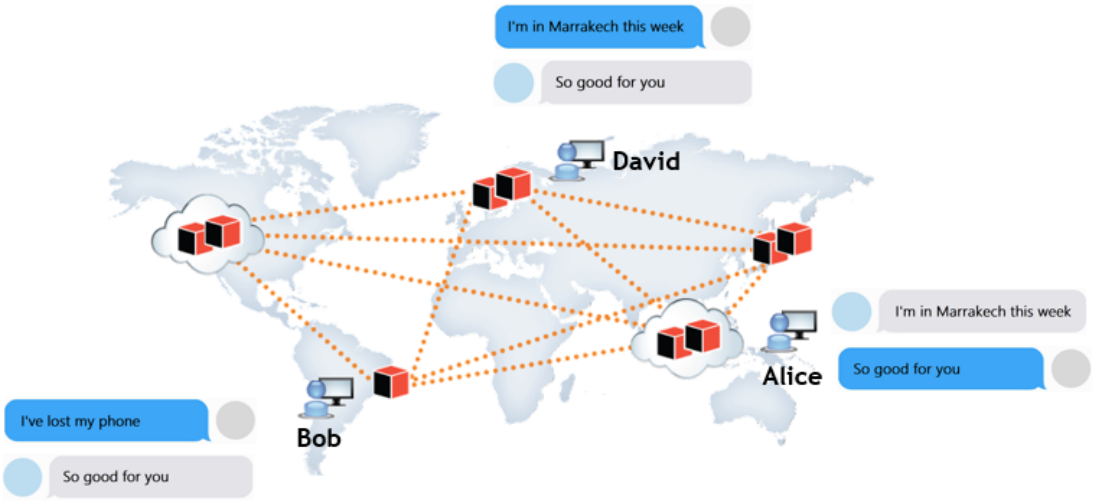


Figure 2.3 – Eventual consistent messages in a messaging application.

Therefore, Bob concludes that Alice message ("So good for you") is an answer to his message ("I've lost my phone"). That is not something that we really want

to happen but it is completely acceptable considering the fact that data is geographically distributed and messages can be delayed. This kind of behaviors are allowed under weak consistency. Contrary to strong consistency which guarantees that all replicas have the same state all time, weak consistency, allows replicas to diverge. Indeed, there exists several variants of weak consistency depending on how replicas divergence is constrained.

2.2.2 CAP Theorem

Almost twenty years ago (in 2000) [42], the conjecture that there is trade-offs between consistency, availability and partition tolerance was introduced by Eric Brewer. This trade-off is known as the CAP-Theorem. Let's first of all explain what each of this notions stands for.

- Consistency: Every read operation should return the last write value.
- Availability: Correct nodes should return a response for all read and write operations in a reasonable amount of time.
- Partition tolerance: The system should continue to operate in presence of network partitions.

Then, the CAP theorem states that, **it is impossible to implement a distributed system that is simultaneously consistent, available, and partition tolerant**.

In the following, we present the consistency models that we consider in this thesis.

2.2.3 Sequential Consistency

The most intuitive model is Sequential Consistency (SC) which is a fundamental model of shared memory formalized by Lamport(1979). "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [47]. In other words, write and read operations are atomic, and operations issued by different threads are interleaved arbitrarily while the order between operations issued by a same thread is preserved. SC offers (to the user of the memory) the strongest consistency guarantees, and therefore the best

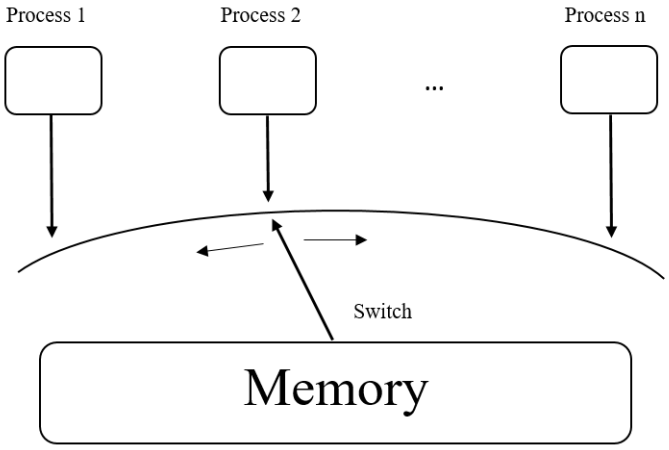


Figure 2.4 – SC architecture.

programming abstraction, since each write operation is considered to be immediately visible to all threads. In terms of memory operations orders, SC maintains all four orders discussed above (*W-R*, *W-W*, *R-W*, *R-R*).

We adopt the formal definition of the Sequential Consistency (SC) model introduced in [12].

Definition 2 *A* history $\langle O, \mathsf{po}, \mathsf{wr} \rangle$ *is* sequentially consistent *if there exists a total relation (called* store order*)* $\mathsf{ww} \subseteq \mathbb{W}(O) \times \mathbb{W}(O)$ *such that the relation* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ *is acyclic.*

The *read-write* relation rw is defined by $\mathsf{rw} = \mathsf{wr}^{-1} \circ \mathsf{ww}$. Intuitively, rw expresses the fact that when a read operation $\mathsf{read}(x, v)$ reads a value v from a write operation $\mathsf{write}(x, v)$, and some other write operation $\mathsf{write}(x, v')$ comes after $\mathsf{write}(x, v)$ in the store order, then there is a conflict between $\mathsf{read}(x, v)$ and $\mathsf{write}(x, v')$, and $\mathsf{read}(x, v)$ must happens before $\mathsf{write}(x, v')$.

Operationally, this formal definition corresponds to an architecture (Figure 2.4) in which there is a global memory and a switch that associates an arbitrary processor to memory at any moment. Each processor issues memory operations in the program order and the switch ensures the total order among all memory operations.

The following examples illustrate the SC definition,

Example 2 *Figure 2.5a shows a* history *that is SC. Since* $\mathsf{read}(y, 0)$ *should precede* $\mathsf{write}(y, 1)$*, this* history *admits a total order where the operations of thread*

t_0:	t_1:
write$(x, 1)$	write$(y, 1)$
read$(y, 0)$	read$(x, 1)$

(a) SC

t_0:	t_1:
write$(x, 1)$	write$(x, 2)$
read$(y, 0)$	read$(y, 0)$
write$(y, 1)$	write$(y, 2)$
read$(x, 1)$	read$(x, 2)$

(b) not SC

Figure 2.5 – Sequential consistency examples.

t_0 are executed before all t_1 thread operations.

Example 3 *Figure 2.5b presents a* history *that does not satisfy SC. The reason is that a total order cannot be found. Since* read$(x, 1)$ *reads the value from* write$(x, 1)$ *and* read$(x, 2)$ *reads the value from* write$(x, 2)$*, all operations of t_0 should be executed before the operations of t_1, or vice versa. This does not allow either t_0 or t_1 to read the value* 0 *on variable y.*

In response to the trade-offs implied by the CAP theorem we have seen above, other weaker memory models are adopted in order to meet performance and/or availability requirements in concurrent/distributed systems. Now, we present some memory models that are weaker than SC, i.e., allow relaxing some orderings (*W-R*, *W-W*, *R-W*, *R-R*). The first one we consider is called Total Store Ordering.

2.2.4 Total Store Ordering

In the Total Store Ordering (TSO) model [60] writes can be delayed, which means that after a write is issued, it is not immediately visible to all threads (except for the thread that issued it), and it is committed later after some arbitrary delay. However, writes issued by the same thread are committed in the same order in which they were issued, and when a write is committed it becomes visible to all the other threads simultaneously. TSO is implemented in hardware but also in a distributed context over a network [43].

The definition of TSO relies on three additional relations: (1) the preserved program order, ppo relation which excludes from the program order pairs formed of a write and respectively, a read operation, i.e., $\mathsf{ppo} = \mathsf{po} \setminus (\mathbb{W}(O) \times \mathbb{R}(O))$, (2) the program order per same location (variable), po-loc relation which is a restriction of po to operations accessing the same variable, i.e., $\mathsf{po\text{-}loc} = \mathsf{po} \cap \{(o, o') \mid \mathsf{var}(o) = \mathsf{var}(o')\}$, and (3) the write-read external relation wr_e which is a

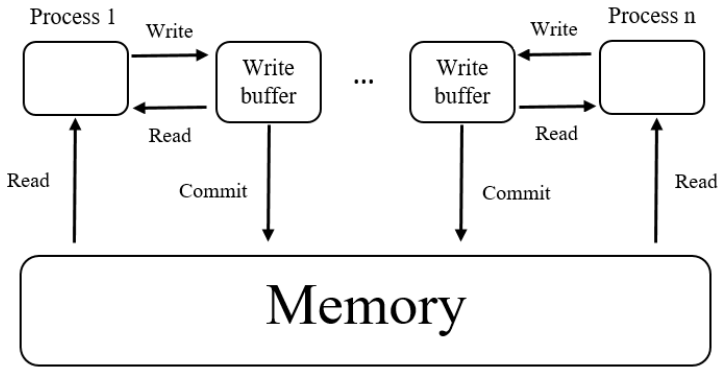


Figure 2.6 – TSO simplified architecture.

restriction of the write-read relation to pairs of operations in different threads (not related by program order), i.e., $\mathsf{wr}_e = \mathsf{wr} \cap \{(o, o') \mid (o, o') \notin \mathsf{po} \text{ and } (o', o) \notin \mathsf{po}\}$. Then,

Definition 3 *A* history $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ *satisfies TSO if there exists a* store order ww *such that the relations* $\mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ *and* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ *are both acyclic.*

Likewise the SC definition, the *read-write* relation is defined by $\mathsf{rw} = \mathsf{wr}^{-1} \circ \mathsf{ww}$.

The formal definition of the TSO model given above is equivalent to the operational model of TSO (See TSO architecture in Figure 2.6) that consists in considering that each thread has a store buffer, and then, each write issued by a thread is first sent to its store buffer before being committed to the memory later in a non-deterministic way. To read a value from some variable x, a thread first checks if a write on x is still pending in its own buffer. In this case it takes the value of this write from the buffer. Otherwise, it fetches the value of x from the memory. Regarding memory operation orderings, TSO model allows the *W-R* order to be violated i.e., write read pairs can be reordered.

To illustrate the TSO definition, consider the following examples,

Example 4 *The Figure 2.7a shows a history which satisfies TSO. The reason is, based on TSO operational model, the operation* $\mathsf{write}(x, 2)$ *of thread* t_1 *can be delayed (pending in the store buffer of* t_1*) until the end of the execution. Therefore, after executing* $\mathsf{read}(z, 0)$*, all the writes of thread* t_0 *are committed to the main memory so that thread* t_1 *can read* 1 *from* y*. Afterwards, it read the value 2 from the variable* x *from its own store buffer.*

t_1:	t_2:
write$(z,1)$	write$(x,2)$
write$(x,1)$	read$(z,0)$
write$(y,1)$	read$(y,1)$
	read$(x,2)$

(a) TSO

t_0:	t_1:	t_2:	t_3:
write$(x,1)$	write$(y,1)$	read$(x,1)$	read$(y,1)$
		read$(y,0)$	read$(x,0)$

(b) not TSO

Figure 2.7 – Total Store Ordering examples.

Example 5 *Figure 2.7b shows a history that is not admitted by TSO. Under TSO, both t_2 and t_3 should see the writes on x and y performed by t_0 and t_1, respectively, in the same order. This is not the case, because t_2 "observes" the write on x before the write on y (since it reads 0 from y) and t_3 "observes" the write on y before the write on x (since it reads 0 from x).*

Other weaker models that impose less constraints on operations ordering are proposed such as causal consistency [46] which is one of the most implemented weak models for distributed systems. Contrary to SC and TSO, causal consistency can be implemented in the presence of faults while ensuring availability.

2.2.5 Causal Consistency

Causal consistency [46] is one of the most used models for replicated objects. It guarantees that if two operations o_1 and o_2 are *causally related* (some process is aware of o_1 when executing o_2), then o_1 should be executed before o_2 in all processes. On the other hand, operations that are not causally related may be seen in different orders by different processes. In the following, we recall the definitions of three causal consistency variations, weak causal consistency, causal convergence and causal memory. We use the same definitions as in [19].

Weak causal consistency

The weakest variation of causal consistency is called *weak causal consistency* (CC, for short). A history is CC if there *exists* a causal order that explains the return value of all operations in the history [19]. Formally,

Definition 4 *A history h satisfies CC w.r.t a specification S if there exists a strict partial order, called* causal order $\mathsf{co} \subseteq O \times O$*, such that, for all operations*

AxCausal	$\mathsf{po} \subseteq \mathsf{co}$
AxArb	$\mathsf{co} \subseteq arb$
AxCausalValue	$\mathsf{CausalHist}(o)\{o\} \preceq \rho_o$
AxCausalSeq	$\mathsf{CausalHist}(o)\{\mathsf{POPast}(o)\} \preceq \rho_o$
AxCausalArb	$\mathsf{CausalArb}(o)\{o\} \preceq \rho_o$

where:
$\mathsf{CausalHist}(o) = (\mathsf{CausalPast}(o), \mathsf{co}, \ell)$
$\mathsf{CausalArb}(o) = (\mathsf{CausalPast}(o), arb, \ell)$
$\mathsf{CausalPast}(o) = \{o' \in O \mid (o', o) \in \mathsf{co}^*\}$
$\mathsf{POPast}(o) = \{o' \in O \mid (o', o) \in \mathsf{po}^*\}$

Table 2.1 – Axioms used in the causal consistency definitions.

$o \in O$ in h, there is a specification sequence $\rho_o \in S$ such that axioms AxCausal *and* AxCausalValue *hold (see Table 2.1).*

Axiom AxCausal states that the causal order should at least include the program order. Axiom AxCausalValue states that, for each operation $o \in O$, a valid sequence of the specification S can be obtained by sequentializing the causal history of o i.e., all operations that precede o in the causal order. In addition, this sequentialization must also preserve the constraints provided by the causal order. Formally, the *causal past* of o, $\mathsf{CausalPast}(o)$, is the *set* of operations that precede o in the causal order. The *causal history* of o, $\mathsf{CausalHist}(o)$, is the restriction of the causal order to the operations in its causal past $\mathsf{CausalPast}(o)$. The notation $\mathsf{CausalHist}(o)\{o\}$ means that only the return value of operation o is kept. The axiom AxCausalValue uses $\mathsf{CausalHist}(o)\{o\}$ because a process is not required to be consistent with the values it has returned in the past or the values returned by the other processes.

The notations $\mathsf{CausalHist}(o)\{o\} \preceq \rho_o$ means that $\mathsf{CausalHist}(o)\{o\}$ can be sequentialized to a sequence ρ_o in the specification. We formally define these last two notations in the next sections.

For a better understanding of this model, consider the following examples.

Example 6 *The history 2.8d is* CC*, we can consider that* $\mathsf{write}(x, 1)$ *is not causally-related to* $\mathsf{write}(x, 2)$*. Therefore, p_2 can execute them in any order.*

Example 7 *The history 2.8e is not* CC*. The reason is that a causal order that explains the return values of all operations in the history cannot be found. Intuitively, since* $\mathsf{read}(y, 1)$ *reads the value from* $\mathsf{write}(y, 1)$*, in any causal order,*

t_1:	t_2:
write$(z,1)$	write$(x,2)$
write$(x,1)$	read$(z,0)$
write$(y,1)$	read$(y,1)$
	read$(x,2)$

(a) CCv (and TSO) but not CM

t_1:	t_2:
write$(x,1)$	write$(x,2)$
read$(x,2)$	read$(x,1)$

(b) CM but not CCv

t_1:	t_2:
write$(x,1)$	write$(x,2)$
read$(y,0)$	read$(y,0)$
write$(y,1)$	write$(y,2)$
read$(x,1)$	read$(x,2)$

(c) CC , CCv and CM

t_1:	t_2:
write$(x,1)$	write$(x,2)$
	read$(x,1)$
	read$(x,2)$

(d) CC but not CCv nor CM

t_1:	t_2:	t_3:
write$(x,1)$	read$(y,1)$	read$(x,2)$
write$(y,1)$	write$(x,2)$	read$(x,1)$

(e) not CC (nor CCv, nor CM)

Figure 2.8 – Causal Consistency examples.

write$(y,1)$ *should precede* read$(y,1)$. *By transitivity of the causal order and because any causal order should include the program order,* write$(x,1)$ *precedes* write$(x,2)$ *in the causal order (*write$(x,1)$ *and* write$(x,2)$ *are causally related). However, process* p_3 *inverse this order. This is a contradiction with the informal definition of CC which requires that every process should see causally related operations in the same order.*

For a better understanding of the weak causal consistency model, let's recall our messaging application example (Figure 2.9). Since Alice message "So good for you " was sent only after reading the David message "I'm in Marrakech this week", these two messages are causally-related. To prevent the previous situation (Figure 2.2), weak causal consistency requires that this two causally-related messages appear in the same order in all replicas. Then, Bob should observes David message "I'm in Marrakech this week" before the Alice message "So good for you".

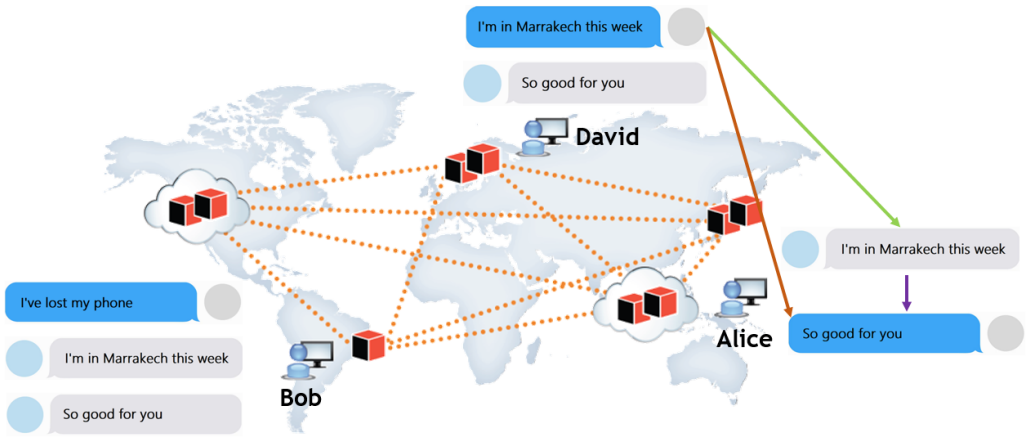


Figure 2.9 – Causally-related messages in a messaging application.

Causal convergence

Causal convergence (CCv, for short) is stronger than CC. It ensures that as long as no new updates are submitted, all processes eventually converge towards the same state. In addition of seeing causally related operations in the same order (CC), causal convergence uses a total order over all the operations in a history to agree on how to order operations which are *not* causally related [19]. This total order is called the *arbitration order* and denoted by *arb*. Similarly to the causal order, the arbitration order is existentially quantified in the CCv definition. Formally,

Definition 5 *A history is CCv w.r.t a specification S if there exists a strict partial order* $\mathsf{co} \subseteq O \times O$ *and a strict total order* $arb \subseteq O \times O$ *such that, for each operation* $o \in O$ *in* h*, there is a specification sequence* $\rho_o \in S$ *such that the axioms* AxCausal*,* AxArb*, and* AxCausalArb *hold.*

Axiom AxArb states that the arbitration order *arb* should at least include the causal order co. Axiom AxCausalArb states that, sequentializing the operations that are in the causal past of o to explain the return value of an operation o, should respect the arbitration order *arb*.

We now present two examples, one which satisfies CCv and another one which violates it.

Example 8 *The history 2.8a is CCv, we can set an arbitration order in which* $\mathsf{write}(x, 1)$ *is ordered before* $\mathsf{write}(x, 2)$*.*

Example 9 *The history 2.8b is not CCv. In order to read* $\mathsf{read}(x, 2)$, $\mathsf{write}(x, 1)$ *must be ordered before* $\mathsf{write}(x, 2)$ *in the arbitration order. On the other hand, to read* $\mathsf{read}(x, 1)$, $\mathsf{write}(x, 2)$ *must be ordered before* $\mathsf{write}(x, 1)$ *in the arbitration order, that is not possible under CCv.*

Causal memory

The third model we consider is *causal memory* (CM, for short) that is also stronger than CC. It guarantees that each process should observe concurrent operations in the same order. In addition, this order should be maintained throughout its whole execution, but it can differ from one process to another [19]. Formally,

Definition 6 *A history* h *is CM w.r.t. a specification* S *if there exists a strict partial order* $\mathsf{co} \subseteq O \times O$ *such that, for each operation* $o \in O$ *in* h*, there is a specification sequence* $\rho_o \in S$ *such that axioms* AxCausal *and* AxCausalSeq *hold.*

Compared to CC, CM requires that each process should be consistent with the return values it has returned in the past. However, a process is not required to be consistent with respect to the return values provided by other processes. Therefore, AxCausalSeq states:

$$\mathsf{CausalHist}(o)\{\mathsf{POPast}(o)\} \preceq \rho_o$$

where $\mathsf{CausalHist}(o)\{\mathsf{POPast}(o)\}$ is the causal history where we only keep the return values of the operations that precede o in the program order (in $\mathsf{POPast}(o)$).

As we have seen above, concurrent values under CCv and CM are required to be observed in the same order by a thread during its entire execution. However, differently from CCv, this order can differ from one thread to another under CM. Although this intuitive description seems to imply that CM is weaker than CCv, the two models are actually incomparable. The following examples illustrate the difference between these models.

Example 10 *The history in Figure 2.8b is allowed by CM, but not by CCv. It is not allowed by CCv because reading 1 from* x *in the first thread implies that it observed* $\mathsf{write}(x, 1)$ *after* $\mathsf{write}(x, 2)$ *while reading 2 from* x *in the second thread implies that it observed* $\mathsf{write}(x, 2)$ *after* $\mathsf{write}(x, 1)$*. While this is allowed by CM where different threads can observe concurrent writes in different orders, it is not allowed by CCv.*

Example 11 *The history in Figure 2.8a is CCv but not CM. It is not allowed by CM because reading the initial value 0 from z implies that* write$(x, 1)$ *is observed after* write$(x, 2)$ *while reading 2 from x implies that* write$(x, 2)$ *is observed after* write$(x, 1)$ *(*write$(x, 1)$ *must have been observed because the same thread reads* 1 *from y and the writes on x and y are causally related). However, under CCv, a thread simply reads the most recent value on each variable and the order in which these values are ordered using timestamps for instance is independent of the order in which variables are read in a thread, e.g., reading* 0 *from z doesn't imply that the timestamp of* write$(x, 2)$ *is smaller than the timestamp of* write$(x, 1)$*. This history is admitted by CCv assuming that the order in which* write$(x, 1)$ *and* write$(x, 2)$ *are observed is* write$(x, 1)$ *before* write$(x, 2)$*.*

Notice that CC is actually strictly weaker that CCv and CM. For instance,

Example 12 *The history in Figure 2.8d is* `CC` *but not* `CCv` *nor* `CM`*. It is* `CC`*, we can consider that* write$(x, 1)$ *is not causally-related to* write$(x, 2)$*. On the other hand, for reading the value* 1 *the thread* t_2 *decides to order* write$(x, 2)$ *before* write$(x, 1)$*, then it changes this order to read the value* 2*. This is not allowed under* `CM` *nor under* `CCv`*.*

The relationship between TSO and CM was not studied yet before. We actually show that this two models are incomparable. To do, consider the following examples.

Example 13 *The history in Figure 2.8a is admitted by TSO, but not by CM (the reasons have been already explained in example 4 and example 11).*

Example 14 *The history in Figure 2.8b is allowed by CM (see example 10), but not by TSO. Since* t_1 *is written in the variable x, it should read the value of x from its own store buffer and read the value 1 not 2. Similarly,* t_2 *is written in the variable x, so it should read the value of x from its own store buffer and read 2 instead of 1.*

Then,

Result 1 *CM and TSO are incomparable.*

The Figure 4.3 summarizes the relationships between the consistency models presented in this chapter. As noticed above, SC is the strongest model we consider in this thesis, TSO is weaker than SC. The causal consistency variants are weaker than TSO and thus than SC.

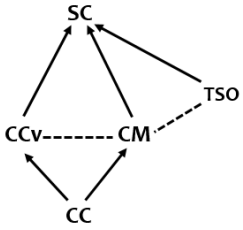


Figure 2.10 – Relationships between consistency models CC, CCv, CM, TSO and SC. Directed arrows denote the "weaker-than" relation while dashed lines connect incomparable models.

2.3 Conclusion

In this chapter, we have introduced the notions and concepts we used through this thesis. We have presented the preliminaries in a first time, then we defined the system model that we consider and the consistency models for which we propose verification methods in the following chapters. The next chapter is dedicated to the verification of causal consistency in its tree variants (CC, CCv and CM).

CHAPTER 3

CAUSAL CONSISTENCY VERIFICATION

As we have seen in the second chapter, the *CAP Theorem* [42] shows that (strong) Consistency, Availability, and Partition tolerance are impossible to be ensured together. In response to this trade-offs implied by the CAP Theorem, weak consistency models were proposed such as causal consistency [46] which is one of the most implemented weak models for distributed systems. Several implementations of different variants of causal consistency (such as causal convergence [50] and causal memory [10]) have been developed i.e., [14, 30, 31, 45, 48, 55, 57]. However, the development of such implementations that meet both consistency requirements and availability and performance requirements is an extremely hard and error prone task. Hence, developing efficient approaches to check the correctness of executions w.r.t consistency models such as causal consistency is crucial. In this chapter we present an approach and a tool for checking automatically the conformance of a system computations to causal consistency. More precisely, we address the problem of, given a computation, how to check its conformance to causal consistency. We consider this problem for three variants of causal consistency that are used in practice and that we have seen in the previous chapter (Chapter 2). Solving this problem constitutes the cornerstone for developing dynamic verification and testing algorithms for causal consistency.

Bouajjani et al. [19] studied the complexity of checking causal consistency for a given computation and showed that it is polynomial time [1]. In addition, they

1. All the causal consistency variations become NP-complete without the assumption that each value is written at most once [19].

formalized the different variations of causal consistency and proposed a reduction of this problem to the occurrence of a finite number of small "bad-patterns" in the computations i.e., some small sets of events occurring in the computations in some particular order. We build on this work in order to define a practical approach and a tool for checking causal consistency, and to apply this tool to real-life case studies. These bad-patterns rely on some relations that we introduce in the next section (3.1) and that can be computed using a least fix point calculation (Datalog programs for instance). Therefore, our approach consists basically in reducing the problem of detecting the existence of these bad patterns in computations to the problem of solving Datalog queries. The fact that solving Datalog queries is polynomial time and that our reduction is polynomial in the size of the computation, allow to solve the conformance checking problem for causal consistency in polynomial time and improves the complexity of this problem from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^3)$. We implement our approach in an efficient testing tool for distributed systems, and carry out several experiments on real distributed databases, showing the efficiency and performance of this approach. To the best of our knowledge, this is the first efficient and full-automated testing tool for causal consistency verification.

Since the experiment results show that CM costs more compared to CC and CCv, we propose a new definition of CM which improves the experiments results of the conformance checking procedure. The new definition of CM computes a small set of constraints compared to the one in [19]. This optimization leads to a better conformance checking approach. This work extends the work originally published in [66, 67, 68].

The rest of this chapter is organized as follows, Section 3.1 recalls the characterization of causal consistency violations (bad-patterns) introduced in [19]. We give new definitions of causal consistency models in Section 3.2. Section 3.3 is dedicated to our causal consistency verification approach. The Subsection 3.3.1 presents the reduction of the problem of conformance checking for causal consistency to the problem of solving Datalog queries. Subsection 3.3.4 describes our testing tool, the case studies we have considered, and the experimental results we obtained.

3.1 Causal consistency violations

In this section, we show, for each causal consistency variant, how to characterize histories that are not conform to it through the presence of some specific sets of operations. In [19], computations that are violations of `CC`, `CCv` or `CM` are characterized by the occurrence of a finite number of particular (small) sets of ordered events, called bad-patterns. Roughly, this characterization describes the small sets of operations that should not occur in some particular order withing a history which satisfies the causal consistency model. We recall these bad-patterns in this section.

Bad-patterns definitions. The tables 3.1 and 3.2 represent the bad-patterns of each causal consistency variant and their definitions respectively.

We now recall the characterization of each causal consistency models based on bad-patterns.

3.1.1 CC Bad-patterns.

We now give the CC bad-patterns as defined in [19]. These bad-patterns are defined using the relation of causality co which is given by the *program order* po or the *write-read* relation wr or any transitive composition of these relations i.e., $\mathsf{co} = (\mathsf{po} \cup \mathsf{wr})^+$.

Lemma 1 *([19]) A history is `CC` if and only if it does not contain any of the bad-patterns `CyclicCO`, `WriteCOInitRead`, `ThinAirRead` and `WriteCORead`.*

To illustrate this, consider the following example.

Example 15 *The history in Figure 2.8e contains the bad-pattern `WriteCORead`, so it is not CC. The* $\mathsf{write}(x,1)$ *is causally ordered before* $\mathsf{write}(x,2)$ *by the transitivity. On the other hand, the process* p_3, $\mathsf{read}(x,1)$ *from* $\mathsf{write}(x,1)$ *(*$(\mathsf{write}(x,1),\mathsf{read}(x,1)) \in \mathsf{wr}$*). The read* $\mathsf{read}(x,1)$ *is also causally-related to* $\mathsf{write}(x,2)$ *by transitivity. The history in Figure 2.8c does not contain any of the bad-patterns, so it is `CC` , `CCv` and `CM`.*

3.1.2 CCv bad-patterns.

As we have seen before, CCv is stronger than CC. Therefore, CCv excludes all the CC bad-patterns we have seen above (`CyclicCO`, `WriteCOInitRead`,

CC	CCv	CM
CyclicCO	CyclicCO	CyclicCO
WriteCOInitRead	WriteCOInitRead	WriteCOInitRead
ThinAirRead	ThinAirRead	ThinAirRead
WriteCORead	WriteCORead	WriteCORead
	CyclicCF	WriteHBInitRead
		CyclicHB

Table 3.1 – Bad-patterns for each causal consistency model

CyclicCO	the causality relation co is cyclic.
WriteCOInitRead	a $\mathsf{read}(x, 0)$ is causally preceded by a $\mathsf{write}(x, v)$ (i.e., $(\mathsf{write}(x, v), \mathsf{read}(x, 0)) \in \mathsf{co}$) such that v $\neq 0$
ThinAirRead	there is a $\mathsf{read}(x, v)$ operation that reads a value v, such that v $\neq 0$, that it is never written before i.e., it can not be related to any write by a wr relation.
WriteCORead	there exist write operations w_1, w_2 such that $\mathsf{var}(w_1) = \mathsf{var}(w_2)$ and a read operation r_1 such that $(w_1, r_1) \in \mathsf{wr}$. In addition, $(w_1, w_2) \in \mathsf{co}$ and $(w_2, r_1) \in \mathsf{co}$.
WriteHBInitRead	there exist a $\mathsf{read}(x, 0)$ and a $\mathsf{write}(x, v)$ (v $\neq 0$) such that $(\mathsf{write}(x, v), \mathsf{read}(x, 0)) \in \mathsf{lhb}_o$ for some operation o, with $(r, o) \in \mathsf{po}^*$.
CyclicHB	the lhb_o relation is cyclic for some operation o.
CyclicCF	the union of cf and co (cf $\cup$ co) is cyclic.

Table 3.2 – Bad-patterns definitions

ThinAirRead and WriteCORead). In addition, CCv excludes another bad pattern, called CyclicCF, defined in terms of a conflict relation cf. Intuitively, two writes w_1 and w_2 on the same variable are in conflict, if w_1 is causally-related to a read taking its value from w_2. Formally, cf is defined as

$$(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{cf} \text{ iff } (\mathsf{write}(x, v), \mathsf{read}(x, v')) \in \mathsf{co} \text{ and } (\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}, \text{ for some } \mathsf{read}(x, v')$$

Then,

Lemma 2 *([19]) A history is* `CCv` *if and only if it is* `CC` *and does not contain the bad-pattern* `CyclicCF`*.*

For instance,

Example 16 *The History in Figure 2.8b is not* `CCv` *as it contains the bad-pattern* `CyclicCF`*. In order to read* $\mathsf{read}(x, 2)$*,* $\mathsf{write}(x, 2)$ *must precedes* $\mathsf{write}(x, 2)$ *in the conflict order. On the other hand, to read* $\mathsf{read}(x, 1)$*,* $\mathsf{write}(x, 2)$ *must be ordered before* $\mathsf{write}(x, 1)$ *in the conflict order. Thus, this leads to CyclicCF bad-pattern.*

3.1.3 CM bad-Patterns.

As we have seen above, CM is also stronger than CC. Therefore, CM excludes all the CC bad-patterns (`CyclicCO`, `WriteCOInitRead`, `ThinAirRead` and `WriteCORead`). In addition, CM excludes two additional bad-patterns (`WriteHBInitRead` and `CyclicHB`), defined using a happened-before relation per operation called lhb_o [2]. Formally, lhb_o is defined as follows.

Definition 7 *Let* $h{=}\langle O, \mathsf{po}, \mathsf{wr}\rangle$ *be a history. For every operation* o *in* h*, let* lhb_o *be the smallest transitive relation such that:*

1. $\mathsf{co}_{|\mathsf{CausalPast}(o)} \subseteq \mathsf{lhb}_o$*, which means that if and only if two operations are causally related and each one is causally related to* o*, then they are related by* lhb_o *i.e.,* $(o_1, o_2) \in \mathsf{lhb}_o$ *if* $(o_1, o_2) \in \mathsf{co}$*,* $(o_1, o) \in \mathsf{co}$ *and* $(o_2, o) \in \mathsf{co}^*$ *(where* co^* *is the reflexive closure of* co*), and*
2. *two writes* w_1 *and* w_2 *are related by* lhb_o *(saturation schema in Fig.3.1c) if* w_1 *is* lhb_o*-related to a read taking its value from* w_2 *and that read is done by the same thread executing* o *and before* o*, i.e.,* $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{lhb}_o$ *if* $(\mathsf{write}(x, v), \mathsf{read}(x, v')) \in \mathsf{lhb}_o$*,* $(\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}$ *and* $(\mathsf{read}(x, v'), o) \in \mathsf{po}^*$ *for some operation* o *(*po^* *is the reflexive closure of po).*

Then,

Lemma 3 *([19]) A history is* `CM` *if and only if it is* `CC` *and does not contain any of the bad-patterns* `WriteHBInitRead` *and* `CyclicHB`*.*

2. This relation was denoted hb_o in [19, 66]. We denote it lhb_o to avoid confusion with other happen-before relations considered in my thesis.

For example,

Example 17 *The history 2.8a contains the bad-pattern* ***WriteHBInitRead*** *so it is not* ***CM****. Let's consider* $lhb = \mathsf{lhb}_{read(x,2)}$*. We have* ($write(z,1)$,$write(x,1)$) $\in$ po *and* ($write(x,1)$, $write(x,2)$) $\in$ *lhb (the reason is that we have* ($write(x,1)$,$read(x,2)$) $\in$ co *and* ($write(x,2)$,$read(x,2)$) $\in$ wr *which implies* ($write(x,1)$,$write(x,2)$) $\in$ lhb*) and* ($write(x,2)$,$read(z,0)$) $\in$ po*, thus by transitivity we have* ($write(z,1)$,$read(z,0)$ $\in$ *lhb.*

In the next section, we present new causal consistency definitions that are based on saturation rules and we show that they are equivalent to the ones in [19], we have seen above.

3.2 New Causal consistency definitions

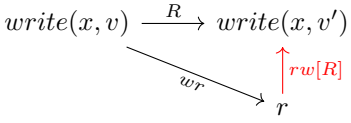
In this section, we present equivalent causal consistency definitions to the bad patterns we have seen above (and thus to the axiomatic definitions used in [19]). These models definitions are based on saturation rules that we are going to see in the next sections. The SC checking approach proposed in Chapter 4 is based on a strong version of these saturation based causal consistency models.

3.2.1 Weak causal consistency

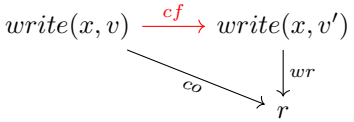
Weak causal consistency (CC) requires that any two causally-dependent values are observed in the same order by all threads, where causally-dependent means that either those values were written by the same thread (i.e., the corresponding writes are ordered by po), or that one value was written by a thread after reading the other value (the wr relation), or any transitive composition of such dependencies, i.e., $\mathsf{co} = (\mathsf{po} \cup \mathsf{wr})^+$. Values written concurrently by two threads can be observed in any order, and even-more, this order may change in time. Formally,

Definition 8 *A* history $\langle O, \mathsf{po}, \mathsf{wr} \rangle$ *satisfies CC if the relation* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *is irreflexive.*

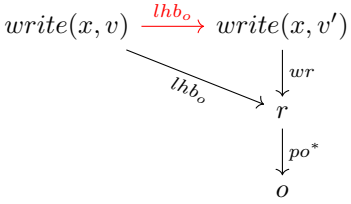
Where the "?" exponent denotes the reflexive closure and ";" is the standard composition. Note that "irreflexive" is used here instead of "acyclic" to say that $\mathsf{rw}[\mathsf{co}]$ is used at most once in the cycle.



(a) Read-Write relation rw[R]



(b) Conflict order cf



(c) Local happen-before lhb_o

Figure 3.1 – Definitions of relations used to define causal consistency models.

The *read-write* relation $\mathsf{rw}[\mathsf{co}]$ induced by the causal relation is defined by

$$(\mathsf{read}(x,v),\mathsf{write}(x,v')) \in \mathsf{rw}[\mathsf{co}] \text{ iff } (\mathsf{write}(x,v),\mathsf{write}(x,v')) \in \mathsf{co} \text{ and } (\mathsf{write}(x,v),\mathsf{read}(x,v)) \in \mathsf{wr}, \text{ for some } \mathsf{write}(x,v)$$

The read-write relation $\mathsf{rw}[\mathsf{co}]$ is a variation of rw from the definition of SC/TSO where the store order ww is replaced by the projection of co on pairs of writes. We obtain $\mathsf{rw}[\mathsf{co}]$ by replacing R in Figure 3.1a by co. In general, given a binary relation R on operations, the read-write relation $\mathsf{rw}[R]$ is defined using the saturation rule in Figure 3.1a as follows:

Definition 9 *The read-write relation* $\mathsf{rw}[R]$ *induced by a relation* R *is defined by* $\mathsf{rw}[R] = \mathsf{wr}^{-1} \circ R_{\mathbb{WW}}$.

We prove that this CC definition is equivalent to the CC bad-patterns we have seen in the first section (Lemma 1).

Proof 1 *($\Rightarrow$) Let* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ *be a history that does not satisfy CC. The lemma 1 implies that h contains one of the bad-patterns CyclicCO, WriteCOInitRead, ThinAirRead and WriteCORead. Let's show that* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *is not irreflexive.*

- *If h contains the CyclicCO bad pattern i.e.,* $(\mathsf{po} \cup \mathsf{wr})^+$ *is cyclic then* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *is not irreflexive.*

- *If h contains the WriteCOInitRead bad pattern i.e., there exists a* $\mathsf{read}(x, 0)$ *that is causally preceded by a* $\mathsf{write}(x, v)$ *(*$(\mathsf{write}(x, v)), \mathsf{read}(x, 0) \in \mathsf{co}$*) such that* $v \neq 0$*. Considering the assumption that every* history *includes a write operation* $\mathsf{write}(x, 0)$ *writing the initial value for each variable* x *and these write operations precede all other operations in* po*. We get* $(\mathsf{write}(x, 0),\ \mathsf{write}(x, v)) \in \mathsf{co}$ *and* $(\mathsf{write}(x, 0), \mathsf{read}(x, 0)) \in \mathsf{wr}$ *then* $(\mathsf{read}(x, 0), \mathsf{write}(x, v)) \in \mathsf{rw}[\mathsf{co}]$*. Thus, the relation* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *is not irreflexive.*

- *If h contains the WriteCORead bad pattern i.e., there exist two write operations* w_1*,* w_2 *such that* $\mathsf{var}(w_1) = \mathsf{var}(w_2)$ *and a read operation* r_1 *such that* $(w_1, r_1) \in \mathsf{wr}$*,* $(w_1, w_2) \in \mathsf{co}$ *and* $(w_2, r_1) \in \mathsf{co}$*. Given* $(w_1, r_1) \in \mathsf{wr}$ *and* $(w_1, w_2) \in \mathsf{co}$ *we get* $(r_1, w_2) \in \mathsf{rw}[\mathsf{co}]$*. Thus,* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *is not irreflexive.*

- *If h contains the ThinAirRead bad pattern i.e., there is a* $\mathsf{read}(x, v)$ *operation which reads a value* v *(*$v \neq 0$*) that it is never written before (there is no* w *operation such that* $(\mathsf{write}(x, v), \mathsf{read}(x, 0)) \in \mathsf{wr}$*). ThinAirRead reads are excluded by the definition of a history and the write-read relation. Their presence can be detected easily and we assumed that this is done a-priori.*

($\Leftarrow$) Consider a history h such that $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *is not irreflexive. Let's prove that this implies the presence of one of the bad patterns: CyclicCO, WriteCOInitRead or WriteCORead. The* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *relation is not irreflexive implies that the history contains one of the following cycles:*

- *A cycle in* $(\mathsf{po} \cup \mathsf{wr})^+$ *which implies directly the CyclicCO bad pattern.*

- *A cycle in* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *(includes only one* $\mathsf{rw}[\mathsf{co}]*). Following its definition, having a* $\mathsf{rw}[\mathsf{co}]$ *relation means that there exist two write operations* w_1*,* w_2 *such that* $\mathsf{var}(w_1) = \mathsf{var}(w_2)$ *and a read operation* r_1 *such that* $(w_1, r_1) \in \mathsf{wr}$*,* $(w_1, w_2) \in \mathsf{co}$ *and* $(r_1, w_2) \in \mathsf{rw}[\mathsf{co}]$*. In order to have a cycle in* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *which includes only one* $\mathsf{rw}[\mathsf{co}]$ *relation, the* (w_2, r_1) *should be in* co *(*$(w_2, r_1) \in \mathsf{co}$*). Thus, we get the WriteCORead bad pattern.*

□

Now let's consider a history that is valid following the new CC definition and another which is not.

Example 18 *The history 2.8d is CC, we can consider that* $\mathsf{write}(x,1)$ *and* $\mathsf{write}(x,2)$ *are not related by the causal order i.e.,* $(\mathsf{write}(x,1), \mathsf{write}(x,2)) \notin \mathsf{co}$. *Therefore, they can be seen in any order by different threads.*

Example 19 *The history 2.8e is not CC. The reason is that we have* $(\mathsf{write}(x,1), \mathsf{write}(x,2)) \in \mathsf{co}$ *by the transitivity which include* $(\mathsf{write}(x,1), \mathsf{write}(y,1)) \in \mathsf{po}$ *and* $(\mathsf{write}(y,1), \mathsf{read}(y,1)) \in \mathsf{wr}$ *and* $(\mathsf{read}(y,1), \mathsf{write}(x,2)) \in \mathsf{po}$. *In* t_3, *we have* $(\mathsf{read}(x,2), \mathsf{read}(x,1)) \in \mathsf{po}$ *then* $(\mathsf{read}(x,1), \mathsf{write}(x,2)) \in \mathsf{rw}[\mathsf{co}]$ *which implies that* $\mathsf{co};(\mathsf{rw}[\mathsf{co}])^?$ *is not irreflexive. Therefore, the history is not allowed by CC.*

3.2.2 Causal convergence

CCv ensures that concurrent values are observed in the same order by all threads. The definition of CCv is based on the conflict order cf we have seen. Then,

Definition 10 *A history* $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ *satisfies CCv if it satisfies CC and the relation* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{cf}$ *is acyclic.*

Let's show that this CCv definition is equivalent to the CCv bad-patterns (Lemma 2).

Proof 2 *(⇒) This is a direct consequence of lemma 2. Let* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ *be a history that does not satisfy CCv. The lemma 2 implies that* h *is not CC or it contains the CyclicCF bad pattern. If it is not CC, then it is not CCv as well. Now, if it contains the CyclicCF bad pattern i.e.,* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{cf}$ *is cyclic then we are done.*

(⇐) If $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{cf}$ *is cyclic, then the history* h *contains the CyclicCF bad pattern (Table 3.2).*

We now present two examples to illustrate the new CCv definition,

Example 20 *The history 2.8a is CCv, we can set a conflict order in which* $\mathsf{write}(x,1)$ *is ordered before* $\mathsf{write}(x,2)$, *so* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{cf}$ *is acyclic.*

Example 21 *The history 2.8b is not CCv. In order to read the value 2,* $\mathsf{read}(x,2)$, *the write* $\mathsf{write}(x,1)$ *must be ordered before* $\mathsf{write}(x,2)$ *in the conflict order* cf. *On the other hand, to read the value 1,* $\mathsf{read}(x,1)$, *the write* $\mathsf{write}(x,2)$ *must be ordered before* $\mathsf{write}(x,1)$ *in the conflict order* cf. *Thus, we get a cycle in* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{cf}$.

3.2.3 Causal Memory

The third model we consider is *causal memory* (CM) which is also a strengthening of CC where roughly, concurrent values are required to be observed in the same order by a thread. In addition, this order should be maintained throughout its whole execution, but it can differ from one thread to another. This is formalized by the happen-before relation per operation lhb_o we have seen in the previous section (definition 7).

Now, we formally define CM.

Definition 11 *A history $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ satisfies CM if it satisfies CC and for each operation o in the history, the lhb_o relation is acyclic.*

Next, we prove that this new CCM definition is equivalent to the CM bad-patterns that we have seen above (Lemma 3).

Proof 3 *($\Rightarrow$) Let $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ be a history that does not satisfy CM. The lemma 3 implies that h is not CC or it contains the bad pattern WriteHBInitRead or CyclicHB. If it is not CC, then it is not CM as well. Now,*

- *If h contains the CyclicHB bad pattern i.e., the lhb_o relation is cyclic for some operation o, then it is not CM (the definition 11).*
- *If h contains the WriteHBInitRead bad pattern i.e., there exists a $\mathsf{read}(x, 0)$ and a $\mathsf{write}(x, v)$ $(v \neq 0)$ such that $(\mathsf{write}(x, v), \mathsf{read}(x, 0)) \in \mathsf{lhb}_o$ for some operation o. As we said before, we assume that every* history *includes a write operation $\mathsf{write}(x, 0)$ writing the initial value for each variable x and these write operations precede all other operations in the program order* po *($(\mathsf{write}(x, 0),\ \mathsf{write}(x, v)) \in \mathsf{po}$ for all $\mathsf{write}(x, v)$). Given $(\mathsf{write}(x, v), \mathsf{read}(x, 0)) \in \mathsf{lhb}_o$ and $(\mathsf{write}(x, 0), \mathsf{read}(x, 0)) \in \mathsf{wr}$, we get $(\mathsf{write}(x, v),\ \mathsf{write}(x, 0)) \in \mathsf{lhb}_o$. Since $(\mathsf{write}(x, 0),\ \mathsf{write}(x, v)) \in \mathsf{po} \subseteq \mathsf{lhb}_o$, we get a cycle in lhb_o.*

($\Leftarrow$) If for an operation o in the history, the lhb_o relation is cyclic, then we get the CyclicHB bad pattern (Table 3.2).

In the following section, we propose an improved causal memory definition (definition 13) alternate to the definition we have seen above (definition 11).

An improved Causal Memory definition

In this section, we propose a succinct but equivalent CM definition (definition 13) which only asks lhb_o to be acyclic for a small set of operations o. As we will see in experiments (Section 3.3.4), this improves the verification runtime. Let's call CM_1 the definition 12 and CM_2 the improved `CM` definition that we propose.

Let's recall the definition that we called CM_1,

Definition 12 *A history $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ satisfies CM_1 if it satisfies `CC` and for each operation o in the history, the relation lhb_o is acyclic.*

Then, CM_2 is defined as follows,

Definition 13 *A history $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ satisfies CM_2 if it satisfies `CC` and for each po-maximal operation o in the history, the relation lhb_o is acyclic.*

To prove the equivalence between definition 12 and 13, we have to prove some intermediate results. First, we define lhb^i_o to denote a controlled saturated version of lhb_o.

Definition 14 *For every operation o in h,*

1. *let lhb^0_o be the relation such that if two operations are causally related and each one is causally related to o, then they are related by lhb^0_o i.e., $(o_1, o_2) \in \mathsf{lhb}_o$ if and only if $(o_1, o_2) \in \mathsf{co}$, $(o_1, o) \in \mathsf{co}$ and $(o_2, o) \in \mathsf{co}^*$ (where co^* is the reflexive closure of co),*
2. *let lhb^i_o for $i > 0$ be the transitive relation if two writes w_1 and w_2 are related by lhb^i_o if w_1 is $(\cup_{j<i}\mathsf{lhb}^j_o)^+$ (transitive closure of all the previous lhb^j_o) related to a read taking its value from w_2 and that read is done by the same thread executing o and before o, i.e., $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{lhb}^i_o$ if and only if $(\mathsf{write}(x, v), \mathsf{read}(x, v')) \in (\cup_{j<i}\mathsf{lhb}^j_o)^+$, $(\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}$ and $(\mathsf{read}(x, v'), o) \in \mathsf{po}^*$ for some $\mathsf{read}(x, v')$.*

Theorem 1 *For all o, $\mathsf{lhb}_o = (\cup_{i\geq 0}\mathsf{lhb}^i_o)^+$*

Proof 4 *By construction, $(\cup_{i\geq 0}\mathsf{lhb}^i_o)^+$ satisfies definition 7. Because lhb_o is the smallest one, $\mathsf{lhb}_o \subseteq (\cup_{i\geq 0}\mathsf{lhb}^i_o)^+$.*

Also, by construction, all the relations in $(\cup_{i\geq 0}\mathsf{lhb}^i_o)^+$ must be present in lhb_o because they are constructed statically from co and wr. So $\mathsf{lhb}_o \supseteq (\cup_{\geq}\mathsf{lhb}^i_o)^+$.

Now, we prove that lhb_o is included in $\mathsf{lhb}_{o'}$ if o is executed before o' in a same thread ($(o, o') \in po$). So, checking lhb_o acyclicity for only po-maximal operations is enough to decide for all operations. To prove this, we use the lhb_o^i definition.

Lemma 4 *If* $(o, o') \in \mathsf{po}$ *then* $\mathsf{lhb}_o^i \subseteq \mathsf{lhb}_{o'}^i$ *for* $i \geq 0$

Proof 5 *The proof is by induction on the index* i *of* lhb_o^i.

- *Base case.* $i = 0$. *Since* $(o, o') \in \mathsf{po} \subseteq \mathsf{co}$, $(o_1, o) \in \mathsf{co}$ *and* $(o_2, o) \in \mathsf{co}^*$ *implies,* $(o_1, o') \in \mathsf{co}$ *and* $(o_2, o') \in \mathsf{co}^*$. *Thus, we get* $\mathsf{lhb}_o^0 \subseteq \mathsf{lhb}_{o'}^0$.
- *Inductive step. If there exists two writes* $\mathsf{write}(x, v), \mathsf{write}(x, v')$ *and a read* $\mathsf{read}(x, v')$ *with* $(\mathsf{write}(x, v), \mathsf{read}(x, v')) \in (\cup_{j<i} \mathsf{lhb}_o^j)^+$ *and* $(\mathsf{read}(x, v'), o) \in \mathsf{po}^*$ *to force* $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{lhb}_o^i$ *relation, then it is also true that* $(\mathsf{write}(x, v), \mathsf{read}(x, v') \in (\cup_{j<i} \mathsf{lhb}_{o'}^j)^+$ *(induction hypothesis) and* $(\mathsf{read}(x, v'), o') \in \mathsf{po}^*$. *Then,* $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{lhb}_{o'}^i$ *is forced as well. Finally,* $\mathsf{lhb}_o^i \subseteq \mathsf{lhb}_{o'}^i$.

Corollary 1 *If* $(o, o') \in \mathsf{po}$ *then* $\mathsf{lhb}_o \subseteq \mathsf{lhb}_{o'}$.

Proof 6 *Direct consequence of theorem 1 and lemma 4.*

Finally, we can prove the equivalence between two `CM` definitions. Both of the definitions requires the history to be `CC`. So, we just need to do it for the acyclicity of lhb_o.

Definition 12 requires lhb_o to be acyclic for all o, whereas definition 13 requires lhb_o to be acyclic for a subset of operations o. So, trivially definition 12 implies definition 13.

For the other direction, we use corollary 1. If $(o, o') \in \mathsf{po}$ then $\mathsf{lhb}_o \subseteq \mathsf{lhb}_{o'}$. Hence, a cycle in lhb_o for some o (if o is po-maximal operation, then we are done) will be also present in $\mathsf{lhb}_{o'}$ for the po-maximal o' because $(o, o') \in \mathsf{po}$.

Theorem 2 *Definition 12 and definition 13 for* `CM` *are equivalent.*

The next section presents our approach for checking the causal consistency models we have seen above.

3.3 Causal Consistency verification

This section presents our causal consistency verification approach.

3.3.1 Reduction to Datalog queries solving

In this section, we introduce our reduction of the problem of checking whether a given computation is a CC, CCv or CM violation to the problem of Datalog queries solving. Datalog is a logic programming language that does not allow functions as predicate arguments. The advantage of using Datalog is that it provides a high level language for naturally defining constraints on relations and that solving Datalog queries is polynomial time [61].

Datalog

A rule in Datalog is a statement of the following form:

$$r_1(v_1) \text{ :- } r_2(v_2), ..., r_i(v_i)$$

Where i$\geq$ 1, r_i are the names of predicates (relations) and v_i are arguments. A Datalog program is a finite set of Datalog rules over the same schema [9, 22]. The left hand side (LHS) is called the rule head and represents the outcome of the query, while the right hand side (RHS) is called the rule body.

Example 22 *For instance, this Datalog program computes the transitivity closure of a given graph (edges are the inputs).*

```
trans(X,Y) :- edge(X,Y).
trans(X,Y)  :-  trans(X,Z), trans(Z,Y).
```

Where the fact edge(a,b) means that there exists a direct edge from a to b.

Datalog and logic programming have some similarities. However, the main difference between them is that logic programming allows using function, but Datalog does not. The next table presents some Datalog notations, that we are going to use in the following sections, and their logical counterparts.

Logical formulas	Datalog formulas
T(x,y)	T(x,y) ⟵
T(x,y) $\vee \neg$ R(x,z)$\vee \neg$ T(z,y)	T(x,y) ⟵ R(x,z), T(x,y)
$\neg$ R(x,z)$\vee \neg$ T(z,y)	⟵ R(x,z), T(z,y)

Table 3.3 – Logical notations and their Datalog equivalence

In the literature, there are three definitions for the semantics of Datalog programs, *model theoretic*, *proof-theoretic* and *fixpoint semantics* [9, 22]. In this work, we consider the *fix-point semantics*.

Fix-point semantics.

This approach is based on the fix-point theory. A fixed point of a function $f()$ is an element e from its domain which is mapped by the function to itself i.e., $f(e) = e$. An operator called immediate consequence operator is defined from the Datalog program rules. In fact, this operator is applied repeatedly on existing facts in order to get new ones until getting a fixed point. Doing so gives a constructive definition of Datalog programs semantics.

Histories Encoding

In our approach, extracted relations from a history (po, wr...) are represented as predicates called facts, while the algorithm for fixed point computation is formulated as Datalog recursive relations called inference rules.
We first introduce facts. For instance, consider the fact po(a,b) which represents the program order from the operation a to the operation b (likewise for po(b,c)),

```
po(a,b).
po(b,c).
```

Now, we define the needed relations in our approach.

- *rd(X)*, X is a read operation
- *wrt(X)*, X is a write operation
- *po(X, Y)*, X precedes Y in the program order po.
- *wr(X, Y)*, Y reads the value from a write operation X (wr relation)
- *sv(X, Y)*, the operations X and Y access to the same variable.

Afterwards, we define the inference rules used to generate derived relations. For instance, the following rules states that the causal relation co is derived from po and wr and it is transitive.

```
co(X,Y) :- po(X,Y). % co=(po U wr)
co(X,Y) :- wr(X,Y). % co=(po U wr)
co(X,Z) :- co(X,Y), co(Y,Z). % co is transitive
```

Bad-patterns Encoding

We have expressed all the bad-patterns as Datalog inference rules, except the ThinAirRead bad-pattern that we verify externally. The reason is that it contains an universal quantification over all operations. There exist two kinds of bad-patterns. The first type is related to the existence of a cycle in a relation. For instance, the bad-pattern CyclicCO that is expressed as

```
:- co(X,Y), co(Y,X). % CyclicCO
```

Intuitively, this means that there exist no operations X and Y such that X precedes Y in the causal order and Y also precedes X in the causal order. Since co is transitive, we can simply write it as

```
:- co(X,X). % CyclicCO
```

The second type of bad-patterns is related to the occurrence of a set of operations in some particular order. For instance, WriteCORead is expressed as follows

```
:- co(X,Y), co(Y,Z), wr(X,Z), wrt(X), wrt(Y), rd(Z), sv(X,Y), sv(Y,Z). %
    WriteCORead
```

Intuitively, this means that there exist no write operations X and Y on the same variable and a read operation Z which takes the value from X such that X precedes Y in the causal order and Y precedes Z in the causal order.

CC bad-patterns encoding.

In addition to the `CyclicCO` bad-pattern we have seen above, we show how the other CC bad-patterns are encoded. Consider the following example which presents the Datalog program corresponding to an execution history.

Example 23 *This example represents the history 2.8b Datalog program for checking CC. Given a history, first we extract all the facts (the relations between operations that we defined in Section 3.3.1). Second, we define the inference rules (* co *definition and its transitivity in this example). Third, we encode the*

bad-patterns of the consistency model that we want to check (The CC consistency model in this case). The CC bad-patterns encoding is shown in the last part of the following Datalog program (After `CC bad patterns` *comment). Mention that the* `CC bad patterns` *and their encoding are already explained above.*

```
% Facts
wrt("w(x,1,id0)").
po("w(x,1,id0)","r(x,2,id1)").
sv("r(x,2,id1)","w(x,1,id0)").
sv("w(x,2,id2)","w(x,1,id0)").
sv("r(x,1,id3)","w(x,1,id0)").
rd("r(x,2,id1)").
sv("w(x,1,id0)","r(x,2,id1)").
wr("w(x,2,id2)","r(x,2,id1)").
sv("w(x,2,id2)","r(x,2,id1)").
sv("r(x,1,id3)","r(x,2,id1)").
wrt("w(x,2,id2)").
sv("w(x,1,id0)","w(x,2,id2)").
sv("r(x,2,id1)","w(x,2,id2)").
po("w(x,2,id2)","r(x,1,id3)").
sv("r(x,1,id3)","w(x,2,id2)").
rd("r(x,1,id3)").
wr("w(x,1,id0)","r(x,1,id3)").
sv("w(x,1,id0)","r(x,1,id3)").
sv("r(x,2,id1)","r(x,1,id3)").
sv("w(x,2,id2)","r(x,1,id3)").
initread("r(a,0,ida)").
% Inference rules
co(X,Y) :- po(X,Y). % co=(po U wr)
co(X,Y) :- wr(X,Y). % co=(po U wr)
co(X,Z) :- co(X,Y), co(Y,Z). % Transitivity
% CC bad-patterns
:- co(X,X). % CyclicCO
:- co(X,Y), wrt(X), initread(Y), sv(X,Y). % WriteCOInitRead
:- co(X,Y), co(Y,Z), wr(X,Z), wrt(X), wrt(Y), rd(Z), sv(X,Y), sv(Y,Z).
% WriteCORead
```

We mention that since the bad pattern WriteCOInitRead includes a predicate initread(Y), we add the *initread("r(a,0,ida)")* to the programs that do not contain a read which reads the initial value.

The result of running this Datalog program using the online clingo version [2] is shown in the following. We mention that when a history satisfies the consistency model we check, the Datalog program returns "SATISFIABLE" and has a model which includes the derived relations. On the other hand, when a history does not satisfy the consistency model, the Datalog program returns "UNSATISFIABLE" and has no model.

```
clingo version 5.5.0
Reading from stdin
Solving...
Answer: 1
po("w(x,1,id0)","r(x,2,id1)") po("w(x,2,id2)","r(x,1,id3)")  co("w(x,1,id0)","
    r(x,2,id1)") co("w(x,2,id2)","r(x,1,id3)") co("w(x,2,id2)","r(x,2,id1)")
    co("w(x,1,id0)","r(x,1,id3)") wr("w(x,2,id2)","r(x,2,id1)") wr("w(x,1,id0)
    ","r(x,1,id3)") sv("r(x,2,id1)","w(x,1,id0)") sv("w(x,2,id2)","w(x,1,id0)"
    ) sv("r(x,1,id3)","w(x,1,id0)") sv("w(x,1,id0)","r(x,2,id1)") sv("w(x,2,
    id2)","r(x,2,id1)") sv("r(x,1,id3)","r(x,2,id1)") sv("w(x,1,id0)","w(x,2,
    id2)") sv("r(x,2,id1)","w(x,2,id2)") sv("r(x,1,id3)","w(x,2,id2)") sv("w(x
    ,1,id0)","r(x,1,id3)") sv("r(x,2,id1)","r(x,1,id3)") sv("w(x,2,id2)","r(x
    ,1,id3)") initread("r(a,0,ida)") wrt("w(x,1,id0)") wrt("w(x,2,id2)") rd("r
    (x,2,id1)") rd("r(x,1,id3)")
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.008s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

The Datalog program is "SATISFIABLE" which means that the history does not contain any of the bad-patterns `CyclicCO`, `WriteCOInitRead` and `WriteCORead`. Therefore, it satisfies CC.

Now, let's see how CCv and CM bad-patterns are encoded. Since the CCv/CM bad-patterns include CC bad-patterns, each CCv/CM Datalog program should contain CC bad-patterns which we have already seen above in addition to some other rules that present in the next sections.

CCv bad-patterns encoding.

Following the same logic of the CC case, the CCv bad-patterns are encoded as follows

```
% CCv inference rules
cf(X,Y) :- co(X,Z), wr(Y,Z), wrt(X), sv(X,Y), sv(X,Z). %Conflict order CF
cf(X,Y) :- cf(X,Z), cf(Z,Y). %Transitivity
cfco(X,Y) :- co(X,Y). %cfco= CF U CO, cfco is the union of cf and co.
cfco(X,Y) :- cf(X,Y). %cfco= CF U CO
%CCv bad-pattern
 :- cfco(X,Y), cfco(Y,X). %CyclicCF (CF U CO is acyclic)
```

Intuitively, the `CyclicCF` bad-pattern encoding means that there exist no operations X and Y such that X precedes Y in the conflict order CF and Y also precedes X in CF.

Let's consider an example of CCv Datalog programs (the history 2.8b Datalog program).

```
% Facts
wrt("w(x,1,id0)").
po("w(x,1,id0)","r(x,2,id1)").
sv("r(x,2,id1)","w(x,1,id0)").
sv("w(x,2,id2)","w(x,1,id0)").
sv("r(x,1,id3)","w(x,1,id0)").
rd("r(x,2,id1)").
sv("w(x,1,id0)","r(x,2,id1)").
wr("w(x,2,id2)","r(x,2,id1)").
sv("w(x,2,id2)","r(x,2,id1)").
sv("r(x,1,id3)","r(x,2,id1)").
wrt("w(x,2,id2)").
sv("w(x,1,id0)","w(x,2,id2)").
sv("r(x,2,id1)","w(x,2,id2)").
po("w(x,2,id2)","r(x,1,id3)").
sv("r(x,1,id3)","w(x,2,id2)").
rd("r(x,1,id3)").
wr("w(x,1,id0)","r(x,1,id3)").
sv("w(x,1,id0)","r(x,1,id3)").
sv("r(x,2,id1)","r(x,1,id3)").
```

```
sv("w(x,2,id2)","r(x,1,id3)").
initread("r(a,0,ida)").
% CC inference rules
co(X,Y) :- po(X,Y). % co=(po U wr)
co(X,Y) :- wr(X,Y). % co=(po U wr)
co(X,Z) :- co(X,Y), co(Y,Z). % Transitivity
% CC bad-patterns
:- co(X,X). % CyclicCO
:- co(X,Y), wrt(X), initread(Y), sv(X,Y). % WriteCOInitRead
:- co(X,Y), co(Y,Z), wr(X,Z), wrt(X), wrt(Y), rd(Z), sv(X,Y), sv(Y,Z).
% WriteCORead

% CCv inference rules
cf(X,Y) :- co(X,Z), wr(Y,Z), wrt(X), sv(X,Y), sv(X,Z). %Conflict order CF
cf(X,Y) :- cf(X,Z), cf(Z,Y). %Transitivity
cfco(X,Y) :- co(X,Y). %cfco= CF U CO, cfco is the union of cf and co.
cfco(X,Y) :- cf(X,Y). %cfco= CF U CO
%CCv bad-pattern
 :- cfco(X,Y), cfco(Y,X). %CyclicCF (CF U CO is acyclic)

 clingo version 5.5.0
Reading from stdin
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 0.009s
CPU Time     : 0.000s
```

As we have seen, the history 2.8b is not CCv so its CCv checking Datalog program is "UNSATISFIABLE".

CM bad-patterns encoding.

Similarly to CC and CCv, the CM bad-patterns are encoded as follows:

```
%CM inference rules
hb(X,0,0) :- co(X,0). % hbo is initialized to causal order
hb(X,Y,0) :- hb(Y,0,0), co(X,Y).
```

```
hb(X,Y,O) :- hb(X,Z,O), po(Z,O), wr(Y,Z), wrt(X), sv(X,Y).% hbo definition
hb(X,Y,O) :- hb(X,Z,O), wr(Y,Z), wrt(X), sv(X,Y). % hbo definition
hb(X,Z,O) :- hb(X,Y,O), hb(Y,Z,O). %Transitivity
%CM bad-patterns
:- hb(X,Y,O), wrt(X), sv(X,Y), po(Y,O), initread(Y). %WriteHBInitRead
:- hb(X,Y,O), hb(Y,X,O). %CyclicHB
```

The `WriteHBInitRead` states that there exist a initread(Y) ($\mathsf{read}(x, 0)$) and a wrt(X) ($\mathsf{write}(x, v)$) such that sv(X,Y) (X and Y are in the same variable) and hb(X,Y,O) for some operation O, with po(Y,O) ($(r, o) \in \mathsf{po}^*$). The `CyclicHB` bad-pattern states that there exist no operations X and Y such that X precedes Y in the lhb_o relation for some operation o.

CM_1 and CM_2 are characterized by the same CM bad-patterns described above. The only difference is that for CM_2, we have added a function which identifies the po-maximal operation in each thread. Then, we replace the operation "O" in the CM bad-patterns above by these identified operations (last operation in each thread) instead of replacing it by all read/write operations in the history (for CM_1).

For a better understanding, consider the instantiation of the CM bad-patterns for CM_1 and CM_2.

- For CM_1: we replace "O" by all operations in the history.

```
%CM inference rules

hb(X,w(x,1,id0),w(x,1,id0)) :- co(X,w(x,1,id0)).
hb(X,Y,w(x,1,id0)) :- hb(Y,w(x,1,id0),w(x,1,id0)), co(X,Y).
hb(X,Y,w(x,1,id0)) :- hb(X,Z,w(x,1,id0)), po(Z,w(x,1,id0)), wr(Y,Z), wrt(X),
    sv(X,Y).
hb(X,Y,w(x,1,id0)) :- hb(X,Z,w(x,1,id0)), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,w(x,1,id0)) :- hb(X,Y,w(x,1,id0)), hb(Y,Z,w(x,1,id0)).
%CM bad-patterns
:- hb(X,Y,w(x,1,id0)), wrt(X), sv(X,Y), po(Y,w(x,1,id0)), initread(Y).
:- hb(X,Y,w(x,1,id0)), hb(Y,X,w(x,1,id0)).
%CM inference rules
hb(X,w(x,2,id2),w(x,2,id2)) :- co(X,w(x,2,id2)).
hb(X,Y,w(x,2,id2)) :- hb(Y,w(x,2,id2),w(x,2,id2)), co(X,Y).
hb(X,Y,w(x,2,id2)) :- hb(X,Z,w(x,2,id2)), po(Z,w(x,2,id2)), wr(Y,Z), wrt(X),
    sv(X,Y).
```

```
hb(X,Y,w(x,2,id2)) :- hb(X,Z,w(x,2,id2)), wr(Y,Z),  wrt(X), sv(X,Y).
hb(X,Z,w(x,2,id2)) :- hb(X,Y,w(x,2,id2)), hb(Y,Z,w(x,2,id2)).
%CM bad-patterns
:- hb(X,Y,w(x,2,id2)), wrt(X), sv(X,Y), po(Y,w(x,2,id2)), initread(Y).
:- hb(X,Y,w(x,2,id2)), hb(Y,X,w(x,2,id2)).

%CM inference rules
hb(X,"r(x,2,id1)","r(x,2,id1)") :- co(X,"r(x,2,id1)").
hb(X,Y,"r(x,2,id1)") :- hb(Y,"r(x,2,id1)","r(x,2,id1)"), co(X,Y).
hb(X,Y,"r(x,2,id1)") :- hb(X,Z,"r(x,2,id1)"), po(Z,"r(x,2,id1)"), wr(Y,Z), wrt
    (X), sv(X,Y).
hb(X,Y,"r(x,2,id1)") :- hb(X,Z,"r(x,2,id1)"), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,"r(x,2,id1)") :- hb(X,Y,"r(x,2,id1)"), hb(Y,Z,"r(x,2,id1)").
%CM bad-patterns
:- hb(X,Y,"r(x,2,id1)"), wrt(X), sv(X,Y), po(Y,"r(x,2,id1)"), initread(Y).
:- hb(X,Y,"r(x,2,id1)"), hb(Y,X,"r(x,2,id1)").
%CM inference rules
hb(X,"r(x,1,id3)","r(x,1,id3)") :- co(X,"r(x,1,id3)").
hb(X,Y,"r(x,1,id3)") :- hb(Y,"r(x,1,id3)","r(x,1,id3)"), co(X,Y).
hb(X,Y,"r(x,1,id3)") :- hb(X,Z,"r(x,1,id3)"), po(Z,"r(x,1,id3)"), wr(Y,Z), wrt
    (X), sv(X,Y).
hb(X,Y,"r(x,1,id3)") :- hb(X,Z,"r(x,1,id3)"), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,"r(x,1,id3)") :- hb(X,Y,"r(x,1,id3)"), hb(Y,Z,"r(x,1,id3)").
%CM bad-patterns
:- hb(X,Y,"r(x,1,id3)"), wrt(X), sv(X,Y), po(Y,"r(x,1,id3)"), initread(Y).
:- hb(X,Y,"r(x,1,id3)"), hb(Y,X,"r(x,1,id3)").
```

- For CM_2: we replace "O" by the last operation in each process in the history.

```
%CM inference rules
hb(X,"r(x,2,id1)","r(x,2,id1)") :- co(X,"r(x,2,id1)").
hb(X,Y,"r(x,2,id1)") :- hb(Y,"r(x,2,id1)","r(x,2,id1)"), co(X,Y).
hb(X,Y,"r(x,2,id1)") :- hb(X,Z,"r(x,2,id1)"), po(Z,"r(x,2,id1)"), wr(Y,Z), wrt
    (X), sv(X,Y).
hb(X,Y,"r(x,2,id1)") :- hb(X,Z,"r(x,2,id1)"), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,"r(x,2,id1)") :- hb(X,Y,"r(x,2,id1)"), hb(Y,Z,"r(x,2,id1)").
%CM bad-patterns
:- hb(X,Y,"r(x,2,id1)"), wrt(X), sv(X,Y), po(Y,"r(x,2,id1)"), initread(Y).
:- hb(X,Y,"r(x,2,id1)"), hb(Y,X,"r(x,2,id1)").
%CM inference rules
hb(X,"r(x,1,id3)","r(x,1,id3)") :- co(X,"r(x,1,id3)").
```

```
hb(X,Y,"r(x,1,id3)") :- hb(Y,"r(x,1,id3)","r(x,1,id3)"), co(X,Y).
hb(X,Y,"r(x,1,id3)") :- hb(X,Z,"r(x,1,id3)"), po(Z,"r(x,1,id3)"), wr(Y,Z), wrt
    (X), sv(X,Y).
hb(X,Y,"r(x,1,id3)") :- hb(X,Z,"r(x,1,id3)"), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,"r(x,1,id3)") :- hb(X,Y,"r(x,1,id3)"), hb(Y,Z,"r(x,1,id3)").
%CM bad-patterns
:- hb(X,Y,"r(x,1,id3)"), wrt(X), sv(X,Y), po(Y,"r(x,1,id3)"), initread(Y).
:- hb(X,Y,"r(x,1,id3)"), hb(Y,X,"r(x,1,id3)").
```

Now, let's consider the whole Datalog programs and their running results.

- For CM_1:

```
% Facts
wrt("w(x,1,id0)").
po("w(x,1,id0)","r(x,2,id1)").
sv("r(x,2,id1)","w(x,1,id0)").
sv("w(x,2,id2)","w(x,1,id0)").
sv("r(x,1,id3)","w(x,1,id0)").
rd("r(x,2,id1)").
sv("w(x,1,id0)","r(x,2,id1)").
wr("w(x,2,id2)","r(x,2,id1)").
sv("w(x,2,id2)","r(x,2,id1)").
sv("r(x,1,id3)","r(x,2,id1)").
wrt("w(x,2,id2)").
sv("w(x,1,id0)","w(x,2,id2)").
sv("r(x,2,id1)","w(x,2,id2)").
po("w(x,2,id2)","r(x,1,id3)").
sv("r(x,1,id3)","w(x,2,id2)").
rd("r(x,1,id3)").
wr("w(x,1,id0)","r(x,1,id3)").
sv("w(x,1,id0)","r(x,1,id3)").
sv("r(x,2,id1)","r(x,1,id3)").
sv("w(x,2,id2)","r(x,1,id3)").
initread("r(a,0,ida)").
% Inference rules
co(X,Y) :- po(X,Y).
co(X,Y) :- wr(X,Y).
co(X,Z) :- co(X,Y), co(Y,Z). % Transitivity
% CC bad-patterns
:- co(X,X). % CyclicCO
:- co(X,Y), wrt(X), initread(Y), sv(X,Y). % WriteCOInitRead
:- co(X,Y), co(Y,Z), wr(X,Z), wrt(X), wrt(Y), rd(Z), sv(X,Y), sv(Y,Z).
```

```
% WriteCORead


%CM inference rules
hb(X,w(x,1,id0),w(x,1,id0)) :- co(X,w(x,1,id0)).
hb(X,Y,w(x,1,id0)) :- hb(Y,w(x,1,id0),w(x,1,id0)), co(X,Y).
hb(X,Y,w(x,1,id0)) :- hb(X,Z,w(x,1,id0)), po(Z,w(x,1,id0)), wr(Y,Z), wrt(X),
    sv(X,Y).
hb(X,Y,w(x,1,id0)) :- hb(X,Z,w(x,1,id0)), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,w(x,1,id0)) :- hb(X,Y,w(x,1,id0)), hb(Y,Z,w(x,1,id0)).
%CM bad-patterns
:- hb(X,Y,w(x,1,id0)), wrt(X), sv(X,Y), po(Y,w(x,1,id0)), initread(Y).
:- hb(X,Y,w(x,1,id0)), hb(Y,X,w(x,1,id0)).
% CM inference rules
hb(X,w(x,2,id2),w(x,2,id2)) :- co(X,w(x,2,id2)).
hb(X,Y,w(x,2,id2)) :- hb(Y,w(x,2,id2),w(x,2,id2)), co(X,Y).
hb(X,Y,w(x,2,id2)) :- hb(X,Z,w(x,2,id2)), po(Z,w(x,2,id2)), wr(Y,Z), wrt(X),
    sv(X,Y).
hb(X,Y,w(x,2,id2)) :- hb(X,Z,w(x,2,id2)), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,w(x,2,id2)) :- hb(X,Y,w(x,2,id2)), hb(Y,Z,w(x,2,id2)).
%CM bad-patterns
:- hb(X,Y,w(x,2,id2)), wrt(X), sv(X,Y), po(Y,w(x,2,id2)), initread(Y).
:- hb(X,Y,w(x,2,id2)), hb(Y,X,w(x,2,id2)).
% CM inference rules
hb(X,"r(x,2,id1)","r(x,2,id1)") :- co(X,"r(x,2,id1)").
hb(X,Y,"r(x,2,id1)") :- hb(Y,"r(x,2,id1)","r(x,2,id1)"), co(X,Y).
hb(X,Y,"r(x,2,id1)") :- hb(X,Z,"r(x,2,id1)"), po(Z,"r(x,2,id1)"), wr(Y,Z), wrt
    (X), sv(X,Y).
hb(X,Y,"r(x,2,id1)") :- hb(X,Z,"r(x,2,id1)"), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,"r(x,2,id1)") :- hb(X,Y,"r(x,2,id1)"), hb(Y,Z,"r(x,2,id1)").
%CM bad-patterns
:- hb(X,Y,"r(x,2,id1)"), wrt(X), sv(X,Y), po(Y,"r(x,2,id1)"), initread(Y).
:- hb(X,Y,"r(x,2,id1)"), hb(Y,X,"r(x,2,id1)").
% CM inference rules
hb(X,"r(x,1,id3)","r(x,1,id3)") :- co(X,"r(x,1,id3)").
hb(X,Y,"r(x,1,id3)") :- hb(Y,"r(x,1,id3)","r(x,1,id3)"), co(X,Y).
hb(X,Y,"r(x,1,id3)") :- hb(X,Z,"r(x,1,id3)"), po(Z,"r(x,1,id3)"), wr(Y,Z), wrt
    (X), sv(X,Y).
hb(X,Y,"r(x,1,id3)") :- hb(X,Z,"r(x,1,id3)"), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,"r(x,1,id3)") :- hb(X,Y,"r(x,1,id3)"), hb(Y,Z,"r(x,1,id3)").
%CM bad-patterns
```

```
:- hb(X,Y,"r(x,1,id3)"), wrt(X), sv(X,Y), po(Y,"r(x,1,id3)"), initread(Y).
:- hb(X,Y,"r(x,1,id3)"), hb(Y,X,"r(x,1,id3)").

clingo version 5.5.0
Reading from stdin
Solving...
Answer: 1
po("w(x,1,id0)","r(x,2,id1)") po("w(x,2,id2)","r(x,1,id3)") co("w(x,1,id0)","r
    (x,2,id1)") co("w(x,2,id2)","r(x,1,id3)") co("w(x,2,id2)","r(x,2,id1)") co
    ("w(x,1,id0)","r(x,1,id3)") wr("w(x,2,id2)","r(x,2,id1)") wr("w(x,1,id0)",
    "r(x,1,id3)") sv("r(x,2,id1)","w(x,1,id0)") sv("w(x,2,id2)","w(x,1,id0)")
    sv("r(x,1,id3)","w(x,1,id0)") sv("w(x,1,id0)","r(x,2,id1)") sv("w(x,2,id2)
    ","r(x,2,id1)") sv("r(x,1,id3)","r(x,2,id1)") sv("w(x,1,id0)","w(x,2,id2)"
    ) sv("r(x,2,id1)","w(x,2,id2)") sv("r(x,1,id3)","w(x,2,id2)") sv("w(x,1,
    id0)","r(x,1,id3)") sv("r(x,2,id1)","r(x,1,id3)") sv("w(x,2,id2)","r(x,1,
    id3)") initread("r(a,0,ida)") wrt("w(x,1,id0)") wrt("w(x,2,id2)") rd("r(x
    ,2,id1)") rd("r(x,1,id3)") hb("w(x,2,id2)","r(x,1,id3)","r(x,1,id3)") hb("
    w(x,1,id0)","r(x,1,id3)","r(x,1,id3)") hb("w(x,2,id2)","w(x,1,id0)","r(x
    ,1,id3)") hb("w(x,1,id0)","r(x,2,id1)","r(x,2,id1)") hb("w(x,2,id2)","r(x
    ,2,id1)","r(x,2,id1)") hb("w(x,1,id0)","w(x,2,id2)","r(x,2,id1)")
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.029s
CPU Time     : 0.000s
```

The Datalog program is "SATISFIABLE" so the history satisfies CM_1.

- For CM_2:

```
% Facts
wrt("w(x,1,id0)").
po("w(x,1,id0)","r(x,2,id1)").
sv("r(x,2,id1)","w(x,1,id0)").
sv("w(x,2,id2)","w(x,1,id0)").
sv("r(x,1,id3)","w(x,1,id0)").
rd("r(x,2,id1)").
sv("w(x,1,id0)","r(x,2,id1)").
wr("w(x,2,id2)","r(x,2,id1)").
sv("w(x,2,id2)","r(x,2,id1)").
sv("r(x,1,id3)","r(x,2,id1)").
wrt("w(x,2,id2)").
```

```
sv("w(x,1,id0)","w(x,2,id2)").
sv("r(x,2,id1)","w(x,2,id2)").
po("w(x,2,id2)","r(x,1,id3)").
sv("r(x,1,id3)","w(x,2,id2)").
rd("r(x,1,id3)").
wr("w(x,1,id0)","r(x,1,id3)").
sv("w(x,1,id0)","r(x,1,id3)").
sv("r(x,2,id1)","r(x,1,id3)").
sv("w(x,2,id2)","r(x,1,id3)").
initread("r(a,0,ida)").
% CC inference rules
co(X,Y) :- po(X,Y).
co(X,Y) :- wr(X,Y).
co(X,Z) :- co(X,Y), co(Y,Z). % Transitivity
% CC bad-patterns
:- co(X,X). % CyclicCO
:- co(X,Y), wrt(X), initread(Y), sv(X,Y). % WriteCOInitRead
:- co(X,Y), co(Y,Z), wr(X,Z), wrt(X), wrt(Y), rd(Z), sv(X,Y), sv(Y,Z).
% WriteCORead

% CM inference rules

hb(X,"r(x,2,id1)","r(x,2,id1)") :- co(X,"r(x,2,id1)").
hb(X,Y,"r(x,2,id1)") :- hb(Y,"r(x,2,id1)","r(x,2,id1)"), co(X,Y).
hb(X,Y,"r(x,2,id1)") :- hb(X,Z,"r(x,2,id1)"), po(Z,"r(x,2,id1)"), wr(Y,Z), wrt
    (X), sv(X,Y).
hb(X,Y,"r(x,2,id1)") :- hb(X,Z,"r(x,2,id1)"), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,"r(x,2,id1)") :- hb(X,Y,"r(x,2,id1)"), hb(Y,Z,"r(x,2,id1)").
%CM bad-patterns
:- hb(X,Y,"r(x,2,id1)"), wrt(X), sv(X,Y), po(Y,"r(x,2,id1)"), initread(Y).
:- hb(X,Y,"r(x,2,id1)"), hb(Y,X,"r(x,2,id1)").

% CM inference rules
hb(X,"r(x,1,id3)","r(x,1,id3)") :- co(X,"r(x,1,id3)").
hb(X,Y,"r(x,1,id3)") :- hb(Y,"r(x,1,id3)","r(x,1,id3)"), co(X,Y).
hb(X,Y,"r(x,1,id3)") :- hb(X,Z,"r(x,1,id3)"), po(Z,"r(x,1,id3)"), wr(Y,Z), wrt
    (X), sv(X,Y).
hb(X,Y,"r(x,1,id3)") :- hb(X,Z,"r(x,1,id3)"), wr(Y,Z), wrt(X), sv(X,Y).
hb(X,Z,"r(x,1,id3)") :- hb(X,Y,"r(x,1,id3)"), hb(Y,Z,"r(x,1,id3)").
%CM bad-patterns
:- hb(X,Y,"r(x,1,id3)"), wrt(X), sv(X,Y), po(Y,"r(x,1,id3)"), initread(Y).
```

```
:- hb(X,Y,"r(x,1,id3)"), hb(Y,X,"r(x,1,id3)").

clingo version 5.5.0
Reading from stdin
Solving...
Answer: 1
po("w(x,1,id0)","r(x,2,id1)") po("w(x,2,id2)","r(x,1,id3)") co("w(x,1,id0)","r
    (x,2,id1)") co("w(x,2,id2)","r(x,1,id3)") co("w(x,2,id2)","r(x,2,id1)") co
    ("w(x,1,id0)","r(x,1,id3)") wr("w(x,2,id2)","r(x,2,id1)") wr("w(x,1,id0)",
    "r(x,1,id3)") sv("r(x,2,id1)","w(x,1,id0)") sv("w(x,2,id2)","w(x,1,id0)")
    sv("r(x,1,id3)","w(x,1,id0)") sv("w(x,1,id0)","r(x,2,id1)") sv("w(x,2,id2)
    ","r(x,2,id1)") sv("r(x,1,id3)","r(x,2,id1)") sv("w(x,1,id0)","w(x,2,id2)"
    ) sv("r(x,2,id1)","w(x,2,id2)") sv("r(x,1,id3)","w(x,2,id2)") sv("w(x,1,
    id0)","r(x,1,id3)") sv("r(x,2,id1)","r(x,1,id3)") sv("w(x,2,id2)","r(x,1,
    id3)") initread("r(a,0,ida)") wrt("w(x,1,id0)") wrt("w(x,2,id2)") rd("r(x
    ,2,id1)") rd("r(x,1,id3)") hb("w(x,2,id2)","r(x,1,id3)","r(x,1,id3)") hb("
    w(x,1,id0)","r(x,1,id3)","r(x,1,id3)") hb("w(x,2,id2)","w(x,1,id0)","r(x
    ,1,id3)") hb("w(x,1,id0)","r(x,2,id1)","r(x,2,id1)") hb("w(x,2,id2)","r(x
    ,2,id1)","r(x,2,id1)") hb("w(x,1,id0)","w(x,2,id2)","r(x,2,id1)")
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.011s
CPU Time     : 0.000s
```

The Datalog program is "SATISFIABLE" so the history satisfies CM_2.

CM_2 computes the lhb_o relation for a small set of operations (po-maximal operations) compared to CM_1. As can be seen above, the size of the Datalog program was considerably reduced when we use CM_2 for a small history. Let alone long histories that contains hundreds of operations. The effect of this will be seen in the experimental results (Section 3.3.4).

3.3.2 An algorithm for checking Causal Consistency

Let's name the procedure which implements the reduction we have seen in the previous section REDUC-to-DATALOG. This procedure takes as input a history h and a causal consistency model $\mathcal{M}$ to check, and returns the corresponding Datalog program $\mathcal{D}$. Afterwards, we call another procedure named

DATALOG-SOLVER which verifies whether the obtained Datalog program $\mathcal{D}$ is SATISFIABLE or not.

Theorem 3 *Algorithm 1 returns* `true` *iff the input history h satisfies the causal consistency model M.*

The correctness of this theorem is ensured by the fact that our reduction is a simple and direct encoding of bad patterns in Datalog and these bad-patterns were proven in [19] to capture exactly the causal consistency violations.

Algorithm 1: Checking Causal Consistency algorithm.

Input: A history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ and a causal consistency model `M`
Output: `true` iff h satisfies `M`

```
1 REDUC-to-DATALOG(h, M)
2 if DATALOG-SOLVER(REDUC-to-DATALOG(h, M)) then
3 |   return true;
4 else
5 |   return false;
```

3.3.3 Complexity

The complexity of a Datalog program is $\mathcal{O}(n^k)$ [62], where n is the number of constants in the input data, and k is the maximum number of variables in a clause. As we have seen in the previous section, given a history $\langle O, \mathsf{po}, \mathsf{wr} \rangle$, the maximum number of variables in a rule in our Datalog programs is 3. Thus, the complexity of our approach is $\mathcal{O}(n^3)$, where n is the computation size (the number of operations). Our approach's complexity is better than the one defined in [19] in which the complexity of checking CC, CCv and CM was shown to be $\mathcal{O}(n^5)$.

3.3.4 Experimental Evaluation

We have investigated the efficiency and scalability of our tool (named *CausalC-Checker*) by applying it to two real-life distributed transactional databases, CockroachDB [3] and Galera [4].

Histories generation: The Figure 3.2 presents the general architecture of the testing procedure we used in our experiments.

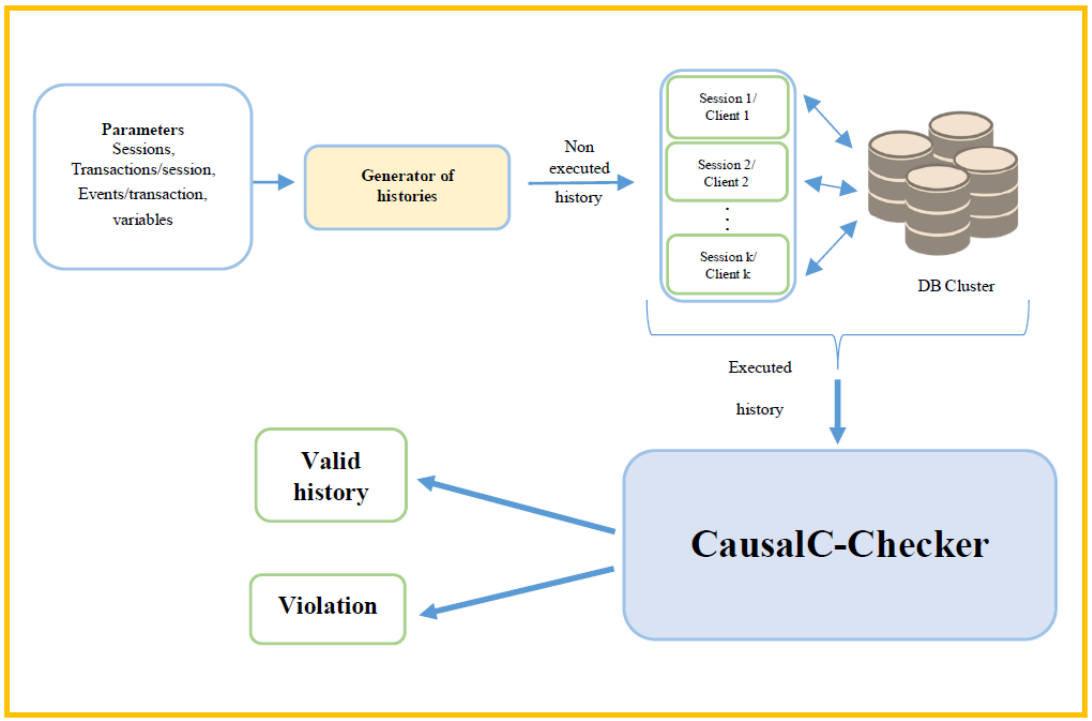


Figure 3.2 – The General architecture of the histories checking procedure

Following the approach used in Jepsen [1], histories are generated using random clients with the parameters, the number of sessions, the number of transactions per session, the number of events per transaction (in this context, we consider one event per transaction), and the number of variables. A client is generated by the generator of histories (Algorithm 2) by choosing randomly the type of operation (read or write) in each transaction, the variable and a value for write operations. This constitutes non executed histories that are the histories which do not contain the return values of read operations. Each client performs a session, communicates with the database cluster by executing operations (read-/write) and gets the return values for read operations. The recorded histories are called executed histories in the Figure 3.2.

We ensure that all histories are differentiated i.e., all written values are unique. These differentiated histories are the input of our *CausalC-Checker* tool.

Case study 1: CockroachDB.

We have used the highly available and strongly consistent distributed database CockroachDB [3] (v2.1.0) that is built on a transactional strongly-

Algorithm 2: The histories generator algorithm

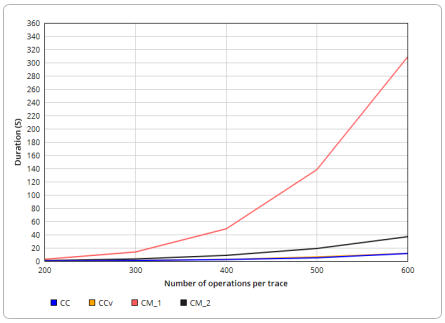
Input: `nClient`, `nTransaction`, `nEvent`, `nVariable`
Output: A non executed history

```
1  lastWrite ← ∅;
2  foreach v ∈ 1..nVariable do
3      lastWrite(v) ← 0;
4  history ← ∅;
5  foreach 1..nClient do
6      Client ← ∅;
7      foreach 1..nTransaction do
8          Transaction ← ∅;
9          foreach 1..nEvent do
10             Event ← new(Event);
11             Event.operation ← uniformly_choose({Read, Write});
12             Event.variable ← uniformly_choose({1..nVariable});
13             if Event.operation = Write then
14                 Event.value ← lastWrite(Event.variable) + 1;
15                 lastWrite(Event.variable) ←
                     lastWrite(Event.variable) + 1;
16             Transaction.push(Event);
17         Client.push(Transaction);
18     history.push(Client);
19 return history;
```

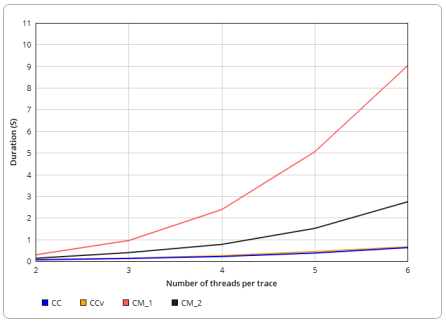
consistent key-value store, so it is expected to be causally consistent. Considering one operation per transaction lead to our model.

We have examined the effect of the number of operations on runtime for a fixed number of processes (4 processes) and the effect of the number of processes. We have tested 200 histories for each configuration and calculated the average runtime.

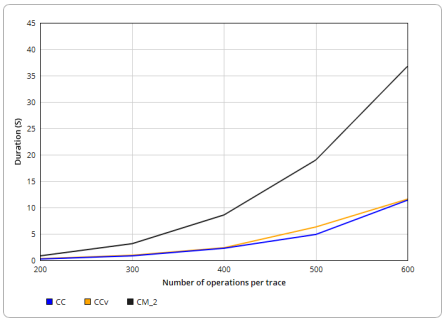
We have checked CC, CCv and CM, using its two definitions CM_1 and CM_2, for all generated histories. The Figure 3.3 shows the results. The graphs 3.3a, 3.3c, 3.3e and 3.3g show the runtime while increasing the number of operations from 100 to 600, in augmentations of 100 (with a fixed number of processes,



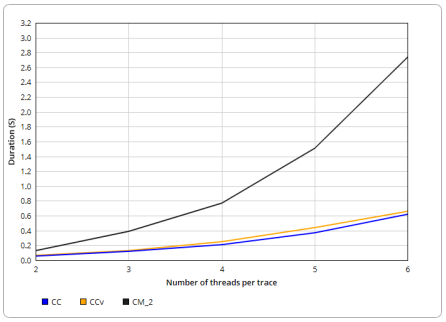
(a) Checking Causal Consistency while varying the number of operations.



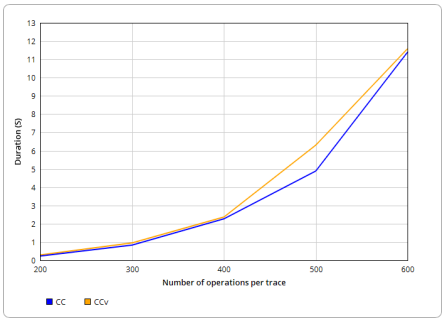
(b) Checking Causal Consistency while varying the number of processes.



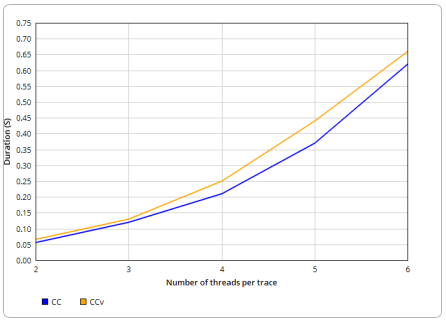
(c) Checking CC, CCv and CM_2 while varying the number of operations.



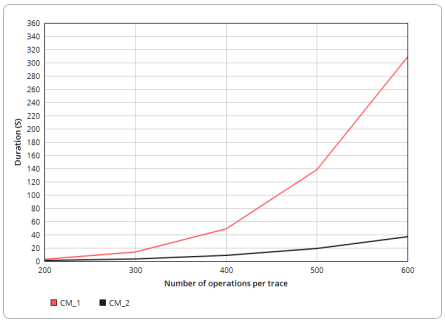
(d) Checking CC, CCv and CM_2 while varying the number of processes.



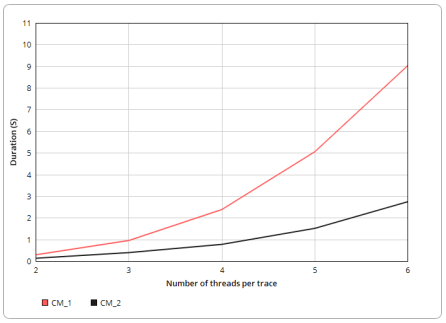
(e) Checking CC and CCv while varying the number of operations.



(f) Checking CC and CCv while varying the number of processes.



(g) Comparing CM_1 and CM_2 runtimes while varying the number of operations.



(h) Comparing CM_1 and CM_2 runtimes while varying the number of processes.

Figure 3.3 – Checking Causal Consistency for CockreachDB histories.

4 processes). The graphs 3.3b , 3.3d, 3.3f and 3.3h report the runtime when increasing the number of processes from 2 to 6, in augmentations of 1. For each number of processes x we have considered $50x$ operations so increasing the number of processes increases the number of operations in the history as well.

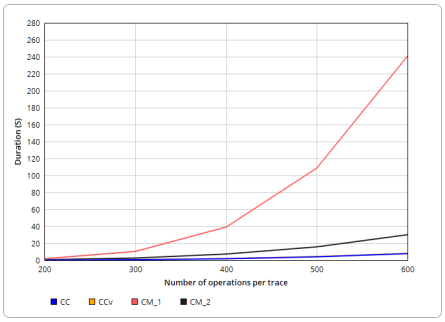
The graph 3.3a resp., 3.3b shows a comparaison between CC, CCv, CM_1 and CM_2 verification runtimes while varying the number of operations resp., the number of processes. The graph 3.3c resp., 3.3d, presents the running time of CM_2 verification compared to CC and CCv verification running time. The graph 3.3e resp., graph 3.3g , shows the evolution of CC and CCv verification resp., CM_1 and CM_2 verification, runtime while increasing the number of operations. The graph 3.3f resp., graph 3.3h, shows the evolution of CC and CCv verification resp., CM_1 and CM_2 verification, runtime while increasing the number of processes.

Our approach is more efficient in the case of CC and CCv verification compared to the CM_1 case (graphs 3.3a and 3.3b). The figure 3.3c resp., 3.3d, is a zoom on CC, CCv and CM_2 of figure 3.3a resp., 3.3b. It shows that the CM_2 improves the running time but costs more compared to CC and CCv as well. The figure 3.3e resp., 3.3f, is a zoom on CC and CCv of figure 3.3a resp., 3.3b. It shows that CC and CCv verification are very efficient and terminates in less than 11.6 seconds for all histories we have tested. As we have noticed above, the results shown in 3.3g and 3.3h show that CM_2 has better performance, by factors of 8 times in the case of 600 operations. As expected, all the tested histories were valid w.r.t. all the considered causal consistency models.

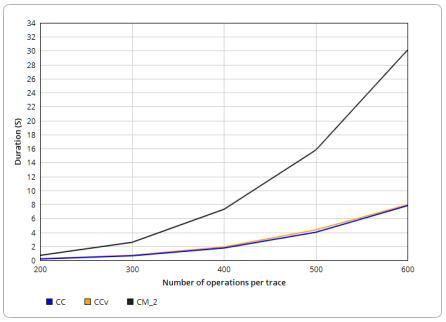
Case study 2: Galera.

We have also used the cluster called Galera [4] (v3.20). Galera Cluster is a database cluster based on synchronous replication and Oracle's InnoDB/MySQL. It is expected to implement *Snapshot isolation* when transactions are processed in separated nodes.
Similarly to the first case study, we have studied the evolution of runtime while increasing the number of operations from 100 to 600, in augmentations of 100. We have verified 200 histories for each number of operations and compute the runtime average.

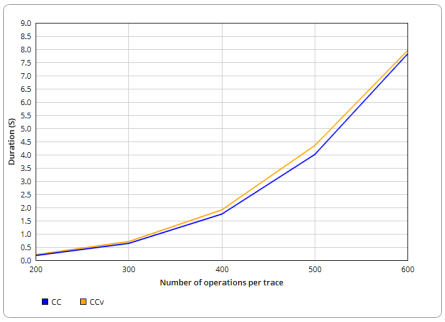
The graphs in Figure 3.4 show the impact of increasing the number of opera-



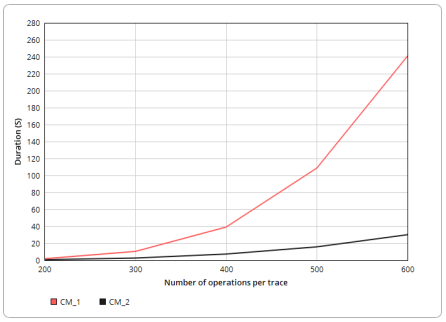
(a) Checking Causal Consistency while varying the number of operations.



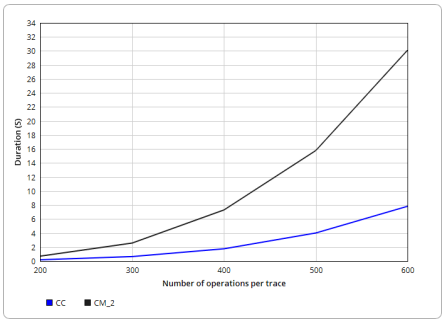
(b) Checking CC, CCv and CM_2 while varying the number of operations.



(c) Checking CC and CCv while varying the number of operations.



(d) Comparing CM_1 and CM_2 runtimes while varying the number of operations.



(e) Comparing CM_2 and CC violations runtimes while varying the number of operations.

Figure 3.4 – Checking Causal Consistency for Galera histories.

tions on runtime while fixing the number of processes (4 processes).

The graph 3.4a shows the comparaison of CC, CCv, CM_1 and CM_2 verification runtimes. The graph 3.4b presents a zoom on graph 3.4a in order to compare CM_2 to CC and CCv. The graph 3.4c reports the evolution of CC and CCv verification runtime. The graph 3.4d shows the evolution of CM_1 and CM_2 checking runtimes. Finally, the graph 3.4e presents a comparison between CC and CM_2 running times.

Likewise the CockroachDB case study, our approach is more efficient in the case of CC and CCv either while increasing the number of operations or processes. The graph 3.4d shows that our new definition CM_2 outperforms CM_1, but still less efficient compared to CC and CCv (graph 3.4b).

Our approach allows capturing violations on the used Galera database. We have found that 1.25% of the tested Galera histories violate causal consistency, that confirms the bugs submitted on Github[5]. We mention that 73.3% of the detected `CM` violations are also `CC` violations. The suggested approach scales well and detects violations on the used version of Galera DB.

The experiments show that our approach is efficient for both verification of valid computations and detection of violations, especially in the case of CC and CCv. The gap between CC (CCv) and CM_1 runtimes reported in the graphs 3.3a, 3.3b and 3.4a is due to the fact that in CM_1 we compute the lhb_o relation and check the bad-patterns for each operation. This gap is reduced using the new definition CM_2 (graphs 3.3a, 3.3b and 3.4a) in which we compute the lhb_o relation and check the bad-patterns for only the last operation of each thread.

3.4 Conclusion

We have presented a tool for checking automatically that given computations of a system are causally consistent (w.r.t CC, CCv and CM). Our procedure for solving this conformance problem is based on implementing the theoretical approach introduced in [19] where causal consistency violations are characterized in terms of the occurrence of some particular bad-patterns. We built on this work by reducing the problem of detecting the existence of these patterns in computations to the problem of solving Datalog queries. Our approach reduces

the complexity of checking CC, CCv and CM from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^3)$. We have applied our algorithm to two real-life case studies. The experimental results have shown that in the case of CC and CCv our approach is efficient and scalable. In the CM case, the costs grow polynomially but much faster than in the case of CC and CCv. In order to improve the CM checking performance, we have proposed an optimized definition (CM_2). Our experimental results have confirmed that this new definition reduces the cost of CM verification and leads to a better conformance checking procedure. However, this optimized CM definition still less efficient compared to CC and CCv.

Since CM_1 and CM_2 cost more compared to CC in terms of runtime (Figures 3.4a and 3.4e) and the most CM violations in practice are CC violations (73.3% in the Galera case), one can start by verifying CC first.

Contrary to causal consistency variants that are checkable in polynomial time under data-independent assumption (As we have mentioned before, all the causal consistency models we have seen are NP-complete without this assumption), there exist other consistency models for which the problem of conformance checking is NP-hard. In the next chapters, we consider the problem of checking some of these models (SC and TSO). The following chapter presents two SC verification approaches based on saturation procedures. The first one uses a strong variant of causal consistency that combines the sets of constraints introduced by CCv and CM.

CHAPTER 4

SEQUENTIAL CONSISTENCY VERIFICATION

Sequential Consistency (SC) [47] is a fundamental model of shared memory, where write and read operations are atomic, and operations issued by different threads are interleaved arbitrarily while the order between operations issued by a same thread is preserved. SC offers (to the user) the strongest consistency guarantees, and therefore the best programming abstraction, since each write operation is considered to be immediately visible to all threads. Other weaker memory models (Causal consistency variants and TSO for instance), adopted in order to ensure performance or availability in some contexts, allow in general write operations to be delayed, not simultaneously visible by all threads, and not being visible in the same order to all threads, which makes programming over such models very hard. However, while adopting SC as a memory model is desirable by memory users as it simplifies their task, it pushes the burden on the memory implementers. Indeed, implementing sequential consistency is extremely complex and error prone due to various optimizations and complex cashing mechanisms that must be adopted in order to achieve acceptable performances.

Therefore, it is highly important to develop automated verification methods and tools for checking SC conformance and detecting subtile bugs in such implementations. A crucial problem for developing bugs detection and testing procedures is checking whether a given execution (of a memory implementation) is SC. However, this problem has been shown to be hard. The reason is that it amounts in finding a total order on write operations that explains the execution,

in the sense that the happen-before relation induced by this order (that includes causality and conflict constraints between writes and reads) is acyclic. It has been shown that this problem is NP-complete in general [38, 41], which means that in the worst case, it is necessary to enumerate the exponentially many possible store orders in order to solve the problem. Therefore, it is very important to investigate methods for solving this problem that avoid falling systematically in the worst case, and that are able to solve it in practice in polynomial time (in the size of the execution) as much as possible. This chapter addresses precisely this issue.

The situation is different for other weaker criteria such as Causal Consistency (CC, CCv and CM). As we have seen in the previous chapter, this models have been shown to be checkable in polynomial time (in the size of the computation) [19, 66]. In fact, causal consistency imposes fewer constraints than SC on the order between writes, and the way it imposes these constraints is "deterministic", in the sense that they can be derived from the history of the execution by applying a least fixpoint computation (which can be encoded for instance, as a standard DATALOG program). All these complexity results hold under the assumption that each value is written at most once, which is without loss of generality for implementations which are data-independent [64], i.e., their behavior doesn't depend on the concrete values read or written in the program. So, any buggy behavior of such implementations can be exposed in executions satisfying this assumption. Notice that as we have seen before, all the causal consistency variants become NP-complete without this assumption. This holds for the variations of the causal consistency we introduce in the next sections as well.

The intrinsic hardness of the problem of checking SC poses a crucial issue for the design of scalable verification or testing techniques for this important consistency model. Tackling this issue requires the development of practical approaches that can work well (with polynomial complexity) when the instance of the problem does not need to generate the worst case (exponential) complexity.

The purpose of this chapter is to propose such an approach. The idea is to reduce the amount of "non-determinism" in searching for the write orders in order to establish SC conformance. For that, our first approach for SC checking is to consider a causal consistency variant CCM (for Convergent Causal Memory),

that is stronger that all known causal consistency variants (CC, CCv and CM), but still weaker than SC, while being polynomial time checkable.

Then, if CCM is already violated by the given computation then we can conclude that the computation does not satisfy the stronger criterion SC. Here the hope is that in practice many computations violating SC can be caught already at this stage using a polynomial time check. Now, in the case that the computation does not violate CCM, we exploit the fact that establishing CCM already imposes a set of constraints on the order between writes. We show that these constraints form a partial order which *must* be a subset of any total write order if it exists, allowing to establish the SC conformance of the computation. Therefore, at this point, it is enough to find an extension of this partial write order, and the hope is that in many practical cases, this set of constraints is already large enough, letting only a small number of pairs of writes to be ordered in order to check SC conformance.

We show experimentally that using CCM allows to improve significantly the performance of SC checking w.r.t. an enumerative approach based on a reduction of the problem to SAT.

Then, a natural question is whether CCM is the strongest model that can be used in this approach? The second section of this chapter considers this question (and some other related questions) and brings answers to them.

We propose a new consistency model called *weak sequential consistency* (wSC, for short) that is defined using a simple *saturation rule* for introducing *store order* constraints. Roughly, the rule applies to a pair of writes; it adds an order constraint between them to avoid a happen-before cycle including a conflict involving one of the writes. Compared to the definition of CCM, the one of wSC is much more natural and simpler. Interestingly, we prove that wSC is strictly stronger than CCM. This is due to the fact that wSC saturation computes a larger set of constraints on pairs of writes than CCM. Then, the question is whether it is possible to do better using a saturation-based definition. In fact we could have considered other saturation rules to define stronger and stronger consistency models approximating SC. But what our experiments show is that the benefit would not be important w.r.t. what is already achieved with wSC. The work presented in this chapter is published in [65, 69].

The rest of this chapter is structured as follows, Section 4.1 presents our first

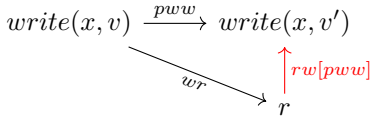
SC checking approach which is based on using CCM as an upper-approximation of SC. We first define the CCM memory model based on saturation rules. Afterwards, we prove some interesting results on the comparison of CCM with existent consistency models. Then, we study the complexity of checking CCM before presenting our algorithm for checking SC. Finally, we evaluate our approach using realistic cache coherence protocols executions. Section 4.2 is dedicated to our second SC checking approach using wSC consistency model. First, we define weak consistency model (wSC). Then, we prove that wSC is stronger than CCM and weaker than SC. Afterwards, we study the complexity of checking wSC followed by a discussion about the notion of SC Kernel. The proposed algorithms for checking SC are presented in Section 4.2.3. Section 4.2.5 describes our testing approach and the obtained experimental results.

4.1 Approach 1: Checking Sequential Consistency Gradually using CCM

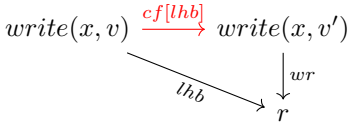
We define an algorithm for checking whether a history satisfies SC which enforces a polynomially-time checkable criterion weaker than SC, a variation of causal consistency, in order to construct a *partial* store order, i.e., one in which not all the writes on the same variable are ordered. This partial store order is then completed until it orders every two writes on the same variable using a standard backtracking enumeration. This approach is efficient when the number of writes that remain to be ordered using the backtracking enumeration is relatively small, a hypothesis confirmed by our experimental evaluation (see Section 4.1.4).

The variation of causal consistency mentioned above, called *convergent causal memory* (CCM, for short), is stronger than existing variations [19] while still being polynomially-time checkable (and weaker than SC). Its definition uses several saturation-based relations between read/write operations which are analogous or even exactly the same relations used to define those variations (CC, CCv and CM).

As we have mentioned, Bouajjani et al. [19] show that the problem of checking whether a history satisfies CC, CCv, or CM is polynomial time. This result is a straightforward consequence of the new definitions of these consistency models that we have introduced in the previous chapter (Section 3.2), since the union



(a) Read-write rw[pww]



(b) Conflict order cf[lhb]

Figure 4.1 – Definitions of relations used to define CCM consistency model.

of relations required to be acyclic can be computed in polynomial time from the relations po and wr which are fixed in a given history. In particular, the union of these relations can be computed by a DATALOG program.

Section 4.1.1 introduces CCM, while Section 4.1.2 presents our algorithm for checking SC. The complexity of our approach is discussed in Section 4.1.3. Finally, Section 4.1.4 presents the experimental results.

4.1.1 Convergent Causal Memory

We define a new variation of causal consistency which builds on causal memory, but similar to causal convergence it enforces that all threads agree on an order in which to observe values written by concurrent (causally-unrelated) writes, and also, it uses a larger read-write relation. Formally,

Definition 15 *A history* $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ *satisfies* convergent causal memory *(CCM) if the relation* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}]$ *is acyclic.*

The *partial store order* pww is defined by

$$\mathsf{pww} = (\mathsf{lhb}_{\mathbb{WW}} \cup \mathsf{cf}[\mathsf{lhb}])^+ \text{ with } \mathsf{lhb} = \Big(\bigcup_{o \in O} \mathsf{lhb}_o\Big)^+.$$

The partial store order pww contains the ordering constraints between writes in all lhb_o relations used to defined causal memory, and also, the conflict relation induced by this set of constraints (a weaker version of conflict relation was used to define causal convergence).

The read-write relation $\mathsf{rw}[pww]$ induced by pww (Figure 4.1a) is defined by $\mathsf{rw}[pww] = \mathsf{wr}^{-1} \circ pww$. The conflict order $\mathsf{cf}[\mathsf{lhb}]$ (Figure 4.1b) is defined by $\mathsf{cf}[\mathsf{lhb}] = \mathsf{lhb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1}$.

t_0:	t_1:
$\mathsf{write}(x,1)$	$\mathsf{write}(x,2)$
$\mathsf{read}(y,0)$	$\mathsf{read}(y,0)$
$\mathsf{write}(y,1)$	$\mathsf{write}(y,2)$
$\mathsf{read}(x,1)$	$\mathsf{read}(x,2)$

(a) CCM but not SC

t_0:	t_1:
$\mathsf{write}(x,1)$	$\mathsf{write}(y,1)$
$\mathsf{write}(x,2)$	$\mathsf{write}(y,2)$
$\mathsf{read}(y,1)$	$\mathsf{read}(y,2)$
	$\mathsf{read}(x,1)$

(b) CM and CCv but not CCM

Figure 4.2 – Histories with two threads used to compare CCM with CC, CCv, CM and SC.

As we have seen in the last chapter, given a *history* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$, for every operation o in h, lhb_o is the smallest transitive relation such that:

- If there exist two operations o_1 and o_2, $(o_1, o_2) \in \mathsf{co}$, and another operation o such that $(o_1, o) \in \mathsf{co}$, and $(o_2, o) \in \mathsf{co}^*$, then $(o_1, o_2) \in \mathsf{lhb}_o$, and
- If there exist two writes $\mathsf{write}(x, v)$ and $\mathsf{write}(x, v')$, a read operation $\mathsf{read}(x, v')$ and another operation o, the lhb_o relation is defined as follows (Fig.3.1c).

$$(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in \mathsf{lhb}_o \text{ if } (\mathsf{write}(x,v), \mathsf{read}(x,v')) \in \mathsf{lhb}_o,$$
$$(\mathsf{write}(x,v'), \mathsf{read}(x,v')) \in \mathsf{wr}, \text{ and}$$
$$(\mathsf{read}(x,v'), o) \in \mathsf{po}^*, \text{ for some } \mathsf{read}(x,v').$$

The following examples show a history which is allowed by CCM and another one which is not.

Example 24 *The Figure 4.2a shows a history that is CCM. The reason is that it admits a partial store order* pww *where the writes in different threads are not ordered i.e., either* t_0 *operations can be executed first followed by* t_1 *operations or the inverse.*

Example 25 *The Figure 4.2b presents a history that does not satisfy CCM. To show this, we use the fact that* pww *relates any two writes which are ordered by program order. Then, we get that* $\mathsf{read}(x,1)$ *and* $\mathsf{write}(x,2)$ *are related by* $\mathsf{rw}[\mathsf{pww}]$ *(because* $\mathsf{write}(x,1)$ *is related by write-read to* $\mathsf{read}(x,1)$*), which further implies that* $(\mathsf{read}(x,1), \mathsf{read}(y,1)) \in \mathsf{rw}[\mathsf{pww}] \circ \mathsf{po}$*. Similarly, we have* $(\mathsf{read}(y,1), \mathsf{read}(x,1)) \in \mathsf{rw}[\mathsf{pww}] \circ \mathsf{po}$*, which implies that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}]$ *is* not *acyclic, so the history does not satisfy CCM.*

As a first result, we show that all the variations of causal consistency presented in the previous chapter, i.e., CC, CCv and CM, are strictly weaker than CCM.

Lemma 5 *If a history satisfies CCM, then it satisfies CC, CCv and CM.*

Proof 7 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ *be a history satisfying CCM. By the definition of* lhb*, we have* $\mathsf{co}_{\mathbb{WW}} \subseteq \mathsf{lhb}_{\mathbb{WW}}$*. In fact, any two writes* o_1 *and* o_2 *related by* co *are also related by* lhb_{o_2}*, which by the definition of* lhb*, implies that they are related by* $\mathsf{lhb}_{\mathbb{WW}}$*. Then, by the definition of* pww*, we have* $\mathsf{lhb}_{\mathbb{WW}} \subseteq \mathsf{pww}$*. This implies that* $\mathsf{rw}[\mathsf{co}] \subseteq \mathsf{rw}[\mathsf{pww}]$ *(by definition,* $\mathsf{rw}[\mathsf{co}] = \mathsf{rw}[\mathsf{co}_{\mathbb{WW}}]$*). Therefore, the acyclicity of* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}]$ *implies that its subset* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{rw}[\mathsf{co}]$ *is also acyclic. The relation* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{rw}[\mathsf{co}]$ *is acyclic implies that the relation* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *is irreflexive.The reason is that, while* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{rw}[\mathsf{co}]$ *excludes cycles with one or more* $\mathsf{rw}[\mathsf{co}]$ *relations,* $(\mathsf{po} \cup \mathsf{wr})^+; (\mathsf{rw}[\mathsf{co}])^?$ *excludes cycles with at most one* $\mathsf{rw}[\mathsf{co}]$ *relation. Thus, h satisfies CC. In addition, it implies that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{cf}[\mathsf{lhb}]$ *is acyclic (the last term of the union is included in* pww*), which by* $\mathsf{co} \subseteq \mathsf{lhb}$*, implies that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{cf}[\mathsf{co}]$ *is acyclic, and thus, h satisfies CCv. The fact that h satisfies CM follows from the fact that h satisfies CC (since* $\mathsf{po} \cup \mathsf{wr}$ *is acyclic) and* lhb *is acyclic (*$\mathsf{lhb}_{\mathbb{WW}}$ *is included in* pww *and the rest of the dependencies in* lhb *are included in* $\mathsf{po} \cup \mathsf{wr}$*).* □

The reverse of the above lemma doesn't hold. To show this, consider the following example.

Example 26 *Figure 4.2b shows a history which satisfies CM and CCv, but it is not CCM. The fact that this history satisfies CM and CCv follows easily from definitions. As we have seen, in CCv and CM concurrent values should be observed in the same order and this order can differ from one thread to another in CM. A possible order for concurrent writes in the variable x is to consider that* $\mathsf{write}(x, 1)$ *precedes* $\mathsf{write}(x, 2)$ *(this is already implied by the* po *order). Similarly, for concurrent values in the variable y,* $\mathsf{write}(y, 1)$ *precedes* $\mathsf{write}(y, 2)$*. Therefore, the threads* t_0 *and* t_1 *can consider this order and thus the history satisfies CCv and CM.*

Then,

Lemma 6 *CCM is strictly stronger than CC, CCv and CM.*

Next, we show that CCM is weaker than SC, which will be important in our algorithm for checking whether a history satisfies SC.

Lemma 7 *If a history satisfies SC, then it satisfies CCM.*

Proof 8 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ *be a history satisfying SC. Then, there exists a* store order ww *such that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}]$ *is acyclic. Let's prove that it satisfies CCM as well. We show that the two relations* $\mathsf{lhb}_{\mathbb{WW}}$ *and* $\mathsf{cf}[\mathsf{lhb}]$*, whose union constitutes* pww*, are both included in* ww*. We first prove that* $\mathsf{lhb} \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+$ *by structural induction on the definition of* lhb_o*,*

1. *If* $(o_1, o_2) \in \mathsf{co} = (\mathsf{po} \cup \mathsf{wr})^+$*, then clearly,* $(o_1, o_2) \in (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+$*,*
2. *If* $(\mathit{write}(x, v), \mathit{read}(x, v')) \in (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+$ *and there is* $\mathit{read}(x, v')$ *such that* $(\mathit{write}(x, v'), \mathit{read}(x, v')) \in \mathsf{wr}$*, then* $(\mathit{write}(x, v), \mathit{write}(x, v')) \in \mathsf{ww}$*. Otherwise, assuming by contradiction that* $(\mathit{write}(x, v'), \mathit{write}(x, v)) \in \mathsf{ww}$*, we get that* $(\mathit{read}(x, v'), \mathit{write}(x, v)) \in \mathsf{rw}[\mathsf{ww}]$ *(by the definition of* $\mathsf{rw}[\mathsf{ww}]$ *using the hypothesis* $(\mathit{write}(x, v'), \mathit{read}(x, v')) \in \mathsf{wr}$*). Note that the latter implies that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}]$ *is cyclic.*

Since $\mathsf{lhb} \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+$*, we get that* $\mathsf{lhb}_{\mathbb{WW}} \subseteq \mathsf{ww}$*. Also, since* $\mathsf{cf}[(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+] \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+$ *(using a similar argument as in point (2) above), we get that* $\mathsf{cf}[\mathsf{lhb}] \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+$*.*

Finally, since $\mathsf{pww} \subseteq \mathsf{ww}$*, we get that* $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}])^+ \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+$*, which implies that the acyclicity of the latter implies the acyclicity of the former. Therefore, h satisfies CCM.* □

The reverse of the above lemma doesn't hold. For instance,

Example 27 *The history in Figure 4.2a is not SC but it is CCM. The reason is that one can consider a partial store order* pww *where the writes of thread* t_0 *are not related to the writes of thread* t_1 *(are not ordered). Since a total order cannot be found, the history is not SC.*

Then,

Lemma 8 *SC is strictly stronger than CCM.*

The Figure 4.3 summarizes the relationships between the consistency models presented above.

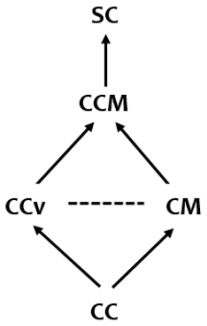


Figure 4.3 – Relationships between consistency models: CC, CCv, CM, CCM and SC.

4.1.2 An Algorithm for Checking Sequential Consistency using CCM

Algorithm 3 checks whether a given history satisfies sequential consistency. As a first step, it checks whether the given history satisfies CCM. If this is not the case, then, by Lemma 7, the history does not satisfy SC as well, and the algorithm returns *false*. Otherwise, it enumerates store orders which extend the partial store order pww, until finding one that witnesses for satisfaction of SC. The history is a violation to SC iff no such store order is found. The soundness of this last step is implied by the proof of Lemma 7, which shows that pww is included in any store order ww witnessing for SC satisfaction.

Algorithm 3: Checking SC conformance: CCM+ENUM algorithm.

Input: A history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$
Output: *true* iff h satisfies SC

```
1 if po ∪ wr ∪ pww ∪ rw[pww] is cyclic then
2     return false;
3 foreach ww ⊃ pww do
4     if po ∪ wr ∪ ww ∪ rw[ww] is acyclic then
5         return true;
6 return false;
```

Theorem 4 *Algorithm 3 returns true iff the input history h satisfies SC.*

4.1.3 Complexity

The partial store order pww includes relations that are defined using saturation rules, then it can be computed in polynomial time (in the size of the input history). Indeed, the lhb_o relations can be computed using a least fixpoint calculation that converges in at most a quadratic number of iterations and acyclicity can be decided in polynomial time. Therefore,

Theorem 5 *Checking whether a history satisfies CCM is polynomial time* ($\mathcal{O}(n^5)$) *in the size of the history.*

4.1.4 Experimental Evaluation

To demonstrate the practical value of the theory developed in the previous sections, we argue that our algorithms are efficient and scalable. We experiment with the SC algorithms we presented above, investigating their running time compared to a standard encoding of these models into boolean satisfiability on a benchmark obtained by running realistic cache coherence protocols within the Gem5 simulator [17] in system emulation mode.

The executions histories that we use in this benchmark are generated using random clients of cache coherence protocols included in the Gem5 distribution. In order to support a memory consistency model, numerous machines provide cache coherence protocols. This protocols ensure that multiple cached copies of data are kept up to date. While Memory models specify the ordering of writes and reads to different memory locations, the cache coherence protocols are guaranteeing a unique order of all writes to the same memory location (the program order). The figure 4.4 presents a simplified memory cache architecture. We used the protocols: MI, MEOSI Hammer, MESI Two Level, and MEOSI AMD Base. Where,

- MI protocol is a simple cache coherence protocol used by default in Gem5. It assumes one-level cache hierarchy, and each node has its own private cache.
- MEOSI protocol: is an implementation of the protocol used in AMD's Hammer chip called AMD's Hammer protocol. It assumes two-level private cache hierarchy L1 and L2 caches, and these caches are private to each node.

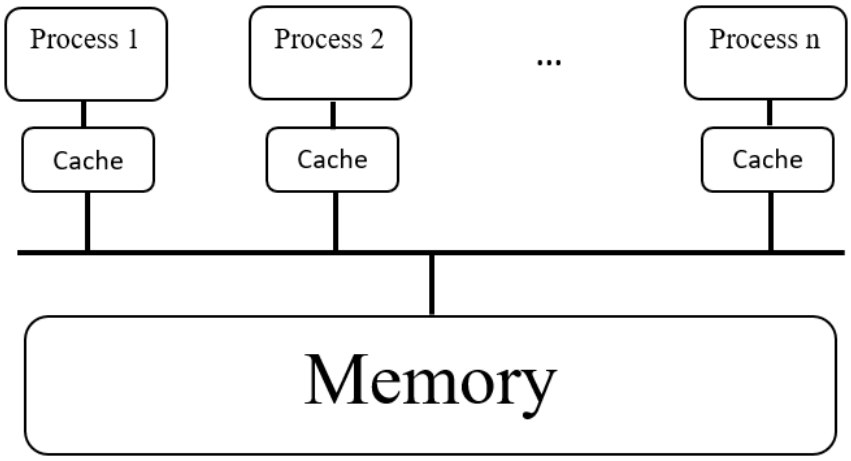


Figure 4.4 – The simplified architecture of a cache memory system

- MESI protocol: is one of the most used cache coherence protocols. It assumes two-level cache hierarchy. L1 is private to each node, while L2 is a shared cache among the nodes.

The randomization process is parametrized by the number of cpus (threads) and the total number of read/write operations. We have actually used the ruby random tester [6]. We ensure that every value is written at most once (data independence assumption).

We have compared two variations of our algorithms for checking SC with a standard encoding of SC into boolean satisfiability (named SC-SAT). The two variations differ in the way in which the partial store order pww dictated by CCM is completed to a total store order ww as required by SC: either using standard enumeration (named SC-CCM+Enum) or using a SAT solver (named SC-CCM+SAT).

The computation of the partial store order pww is done using an encoding of its definition into a DATALOG program. The inductive definition of lhb_o supports an easy translation to DATALOG rules, and the same holds for the union of two relations, or their composition. We used Clingo [39] to run DATALOG programs.

The Figure 4.5 presents the general schema of the algorithm we used in our experiments to check SC using CCM-based approach.

Figure 4.6 reports on the running time of the three algorithms while increasing the number of operations or cpus. All the histories considered in this experiment satisfy SC. This is intended because valid histories force our algorithms to enu-

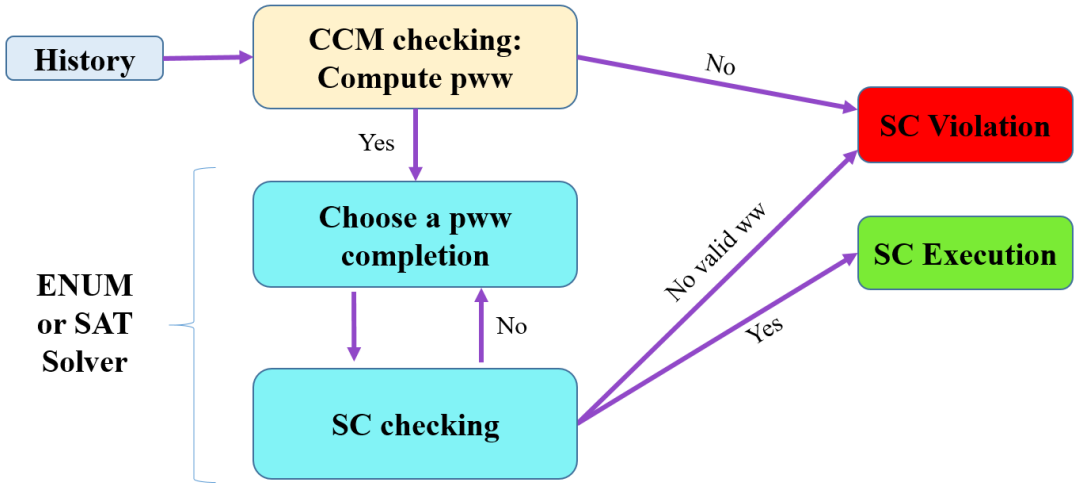
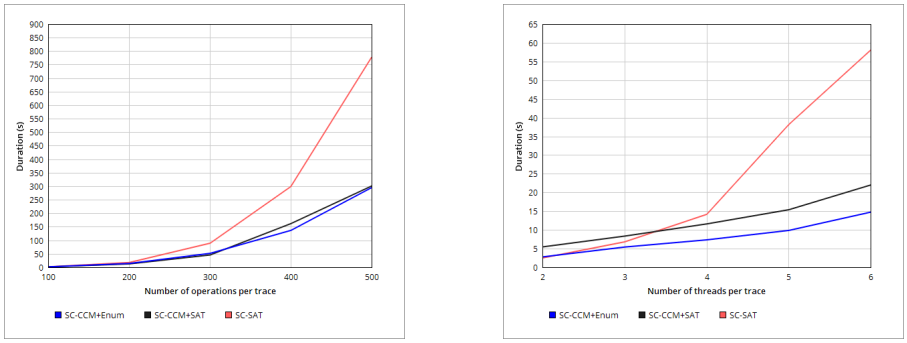


Figure 4.5 – The general schema of the SC checking procedure using CCM

merate extensions of the partial store order (SC violations may be detected while checking CCM). The graph on the left pictures the evolution of the running time when increasing the number of operations from 100 to 500, in increments of 100 (while using a constant number of 4 cpus). For each number of operations, we have considered 200 histories and computed the average running time. The graph on the right shows the running time when increasing the number of cpus from 2 to 6, in increments of 1. For x cpus, we have limited the number of operations to $50x$. As before for each number of cpus, we have considered 200 histories and computed the average running time. As it can be observed, our algorithms scale much better than the SAT encoding and interestingly enough, the difference between an explicit enumeration of pww extensions and one using a SAT solver is not significant. Note that even small improvements on the average running time provide large speedups when taking into account the whole testing process, i.e., checking consistency for a possibly large number of (randomly-generated) executions. For instance, the work on McVerSi [33], which focuses on the complementary problem of finding clients that increase the probability of uncovering bugs, shows that exposing bugs in some realistic cache coherence implementations requires even 24 hours of continuous testing.

Since the bottleneck in our algorithms is given by the enumeration of pww extensions, we have measured the percentage of pairs of writes that are *not* ordered by pww. Thus, we have considered a random sample of 200 histories (with 200



(a) Checking SC while varying the number of operations.

(b) Checking SC while varying the number of cpus.

Figure 4.6 – Checking SC for valid histories.

operations per history) and evaluated this percentage to be just 6.6%, which is surprisingly low. This explains the net gain in comparison to a SAT encoding of SC, since the number of pww extensions that need to be enumerated is quite low. As a side remark, using CCv instead of CCM in the algorithms above leads to a drastic increase in the number of unordered writes. For the same random sample of 200 histories, we conclude that using CCv instead of CCM leaves 57.75% of unordered writes in average which is considerably bigger than the percentage of unordered writes when using CCM.

We have also evaluated our algorithms on SC violations. These violations were generated by reordering statements from the MI implementation, e.g., swapping the order of the actions `s_store_hit` and `p_profileHit` in the transition `transition(M, Store)`. As an optimization, our implementation checks gradually the weaker variations of causal consistency CC and CCv before checking CCM. This is to increase the chances of returning in the case of a violation (a violation to CC/CCv is also a violation to CCM and SC). We have considered 1000 histories with 100 to 400 operations and 2 to 8 cpus, equally distributed in function of the number of cpus. Figure 4.7 reports on the evolution of the average running time. Since these histories happen to all be CCM violations, SC-CCM+Enum and SC-CCM+SAT have the same running time. As an evaluation of our optimization, we have found that 50% of the histories invalidate weaker variations of causal consistency, CC or CCv.

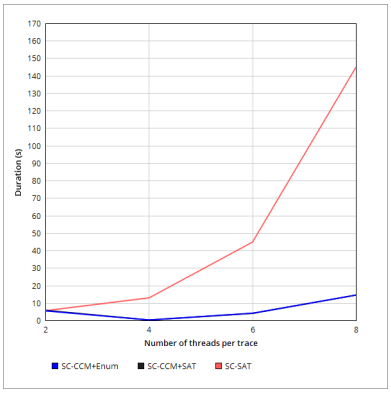


Figure 4.7 – Checking SC for invalid histories while increasing the number of cpus.

4.1.5 Discussion

To summarize, the approach we have presented above (let's call it Gradual Consistency Checking, GCC for short) consists in using weaker consistency models that are known to be polynomially checkable, such as causal consistency, in two ways. First, finding violations for the weaker models allows to detect efficiently some of the SC violations. This can be useful since many violations are already violations for much weaker consistency models. Second, and this is the important point, we used weak consistency models for which checking conformance is based on computing, by a polynomial time fixpoint calculation, a set of order constraints on writes, and these constraints are included in every store order witnessing SC conformance, if any. So, computing these constraints reduces the number of pairs of writes for which an order must be found non-deterministically. We proposed for that a model called Convergence Causal Memory (CCM) that is stronger than all known variants of causal consistency, constructed by combining the constraints imposed by CCv [20] and CM [11, 54]. We have shown experimentally that using CCM allows to improve drastically the performance of SC checking w.r.t. a straightforward enumerative approach (using a reduction of the problem to SAT).

As we have seen in related work (Section 1, Chapter 1), another approach for tackling the issue of efficient SC checking has been introduced recently in [7] and [18]. In these two papers, the authors prove the interesting fact that when the number of threads is fixed, the problem of verifying SC conformance

of a single execution is polynomial time in the size of the execution. For that, they provide algorithms for solving this problem based on clever exploring the interleavings of operations of the execution, exploiting the specific properties of SC. This approach too, let us call it *bounded-thread consistency checking* (BTCC, for short), allows to obtain important performance gains w.r.t. the enumerative approach.

However, several questions can be asked concerning the two approaches described above.

1. As mentioned above, GCC is based on computing a set of constraints on write operations that must be included in all store orders witnessing SC. The computation of these constraints actually uses saturation rules that consist, roughly speaking, in adding store order constraints when their reverse introduces new conflicts between reads and writes such that the induced happen-before relation is cyclic. Saturation-based computations can be done in polynomial time. A natural question is how far this approach can be pushed i.e., is there any stronger consistency model (stronger that CCM and still weaker than SC) that can be used in this approach? This question leads to the following one: Given an execution that is SC, let us call the SC-kernel of this execution the intersection of all store order relations allowing to establish that the execution is SC (i.e., for which the induced happen-before relation is acyclic). Then, is it possible to compute the SC-kernel of any execution using saturation?

2. BTCC exploits in an essential way the fact that the number of threads is fixed. While this approach reduces the complexity in the number of operations, which is important for scalability when the size of executions increases, it does not avoid the fact that the time complexity increases exponentially in the number of threads. So, natural questions is how BTCC scales when both the number of operation and the number of threads increases, and how it compares in this respect with the GCC approach?

3. Another natural question is, since the two approaches use different paradigms, whether it would be useful to combine them to get the best of the two.

The next section explores these questions and brings answers to them.

4.2 Approach 2: Checking Sequential Consistency using wSC

In this section, we introduce a new model called weak SC as a saturation-based approximation of SC. We first prove an interesting result which states that wSC is strictly stronger than CCM and weaker than SC. Then, we show that the wSC saturation rule does not compute the whole SC-kernel in general. We analyze the reason of this by providing several families of counterexamples. We show that there are order constraints that must be imposed on pairs of writes to avoid happen-before cycles including not only one conflict (as wSC saturation does), but several (actually any number) of conflicts involving an arbitrary number of writes. Moreover, we show that in order to impose an order constraint on pairs of writes, in some cases it is necessary to enumerate the possible order of several other pairs of writes, and the number of these pairs can be arbitrarily high. This shows that the design of a saturation-based schema for computing the SC-kernel would require the addition of an unbounded number of saturation rules. This leaves open the theoretical question whether there is a way to compute in polynomial time the SC-kernel of an SC execution).

Nevertheless, even if the wSC saturation does not always capture the SC-kernel, an interesting question is how far is wSC saturation from computing the SC-kernel in practice? We show experimentally that, surprisingly, for executions of real protocols [1], wSC allows to compute the full SC-kernel in most of the cases (more than 74% of the executions), and in general it computes almost the whole SC-kernel (around 99.9% of it). Interestingly, the experiments also show that CCM computes 100% of the SC-kernel for only 0.7% of the executions of the considered benchmark. This shows that the saturation rule we consider for wSC is very powerful and efficient in practice, despite its simplicity (and that it is theoretically not complete as discussed above).

The experimental results, show that wSC leads to a more efficient gradual consistency checking than CCM, and that it scales much better compared to the bounded-thread consistency checking algorithm when the number of threads (and therefore the number of operations as well) increases, while bounded-thread

1. We consider the same 4 protocols from the Gem5 platform that we used in the first section.

consistency checking is in general more efficient for small number of threads. This leads us to define an algorithm using saturation to enhance bounded-thread consistency checking. The obtained algorithm take advantage of both techniques and is shown to be very efficient and scalable.

Section 4.2.1 presents wSC. Section 4.2.2 introduces our results about the SC kernel. Section 4.2.3 presents our algorithms for verifying SC while experimental results are drown in Section 4.2.5.

4.2.1 Weak Sequential Consistency

We define our new consistency model obtained by computing a store order using a simple *saturation rule*. This amounts in using induction in order to define the store order unlike the SC case where it is existentially quantified. Formally, let st and hb be the smallest relations such that

$$\begin{aligned} \mathsf{st} &= ((\mathsf{hb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1}) \cup \mathsf{hb}_{\mathbb{WW}})^{+} \\ \mathsf{hb} &= (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{st} \cup \mathsf{rw}[\mathsf{st}])^{+} \\ \mathsf{rw}[\mathsf{st}] &= \mathsf{wr}^{-1} \circ \mathsf{st} \end{aligned}$$

The $\mathsf{hb}_{\mathbb{WW}}$ is the projection of hb on pairs of writes on the same variable and $\mathsf{hb}_{\mathbb{WR}}$ is the projection of hb on pairs of writes and reads on the same variable.

Then,

Definition 16 *a* history $\langle O, \mathsf{po}, \mathsf{wr} \rangle$ *is conform to weak sequential consistency (wSC) if the* hb *relation is acyclic.*

To illustrate this definition, consider the next examples.

Example 28 *Figure 4.8a shows a* history *which satisfies wSC. To show this, one can consider a partial store order st where the writes* $\mathsf{write}(z, 1)$ *and* $\mathsf{write}(z, 2)$ *are not ordered.*

Example 29 *The Figure 4.8b presents a* history *that does not satisfies wSC. Since* $\mathsf{rw}[st]$ *is included in* hb, $\mathsf{read}(y, 0)$ *is visible to* $\mathsf{write}(y, 2)$ *then* $\mathsf{write}(x, 1)$ *precedes* $\mathsf{read}(x, 2)$ *in* hb. *Thus,* $\mathsf{write}(x, 2)$ *should be executed before* $\mathsf{write}(x, 1)$. *Similarly* $\mathsf{write}(x, 2)$ *precedes* $\mathsf{read}(x, 1)$ *in* hb *as well and* $\mathsf{write}(x, 1)$ *should be executed before* $\mathsf{write}(x, 2)$. *Therefore, we get a cycle in* hb.

t_0:	t_1:	t_2:
$\mathsf{read}(z,2)$	$\mathsf{write}(x,1)$	$\mathsf{write}(t,1)$
$\mathsf{write}(y,2)$	$\mathsf{write}(y,1)$	$\mathsf{write}(s,1)$
$\mathsf{read}(x,1)$	$\mathsf{write}(z,1)$	$\mathsf{write}(z,2)$
t_3:	t_4:	t_5:
$\mathsf{read}(z,2)$	$\mathsf{read}(z,1)$	$\mathsf{read}(z,1)$
$\mathsf{write}(x,2)$	$\mathsf{write}(t,2)$	$\mathsf{write}(s,2)$
$\mathsf{read}(y,1)$	$\mathsf{read}(s,1)$	$\mathsf{read}(t,1)$

(a) wSC but not SC

t_0:	t_1:
$\mathsf{write}(x,1)$	$\mathsf{write}(x,2)$
$\mathsf{read}(y,0)$	$\mathsf{read}(y,0)$
$\mathsf{write}(y,1)$	$\mathsf{write}(y,2)$
$\mathsf{read}(x,1)$	$\mathsf{read}(x,2)$

(b) CCM but not wSC nor SC

Figure 4.8 – Comparison of CCM, wSC and SC consistency models.

We prove that wSC is stronger than CCM (which is already stronger that all known variants of causal consistency).

Lemma 9 *If a* history *satisfies wSC, then it satisfies CCM.*

Proof 9 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ *be a* history *satisfying wSC i.e.,* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{st} \cup \mathsf{rw}[\mathsf{st}]$ *is acyclic. We prove that* $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}])^+ \subseteq \mathsf{hb}$ *(hence the history satisfies also CCM).*

Let us show that for every operation o *in* h, $\mathsf{lhb}_o \subseteq \mathsf{hb}$. *For that, we show that* hb *satisfies the two properties of* lhb_o*:*

- *If* $(o_1, o_2) \in \mathsf{co}$, $(o_1, o) \in \mathsf{co}$, *and* $(o_2, o) \in \mathsf{co}$ *then* $(o_1, o_2) \in \mathsf{hb}$ *trivially holds (since* $\mathsf{co} \subseteq \mathsf{hb}$*), and*
- *If* $(\mathit{write}(x,v), \mathit{read}(x,v')) \in \mathsf{hb}$ *and* $(\mathit{write}(x,v'), \mathit{read}(x,v')) \in \mathsf{wr}$ *then* $(\mathit{write}(x,v), \mathit{write}(x,v')) \in (\mathsf{hb} \circ \mathsf{wr}^{-1})$ *and hence* $(\mathit{write}(x,v), \mathit{write}(x,v')) \in \mathsf{st}$ *and* $(\mathit{write}(x,v), \mathit{write}(x,v')) \in \mathsf{hb}$.

Thus, we have $\mathsf{lhb}_o \subseteq \mathsf{hb}$ *and hence* $\mathsf{lhb} \subseteq \mathsf{hb}$.

Let us now show that $\mathsf{pww} = (\mathsf{lhb}_{\mathbb{WW}} \cup \mathsf{cf}[\mathsf{lhb}])^+ \subseteq \mathsf{st}$. *It is easy to see that* $\mathsf{lhb}_{\mathbb{WW}} \subseteq \mathsf{hb}_{\mathbb{WW}}$ *(since* $\mathsf{lhb} \subseteq \mathsf{hb}$*). By definition, we have also that* $\mathsf{cf}[\mathsf{lhb}] = (\mathsf{lhb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1})$ *and hence* $\mathsf{cf}[\mathsf{lhb}] \subseteq (\mathsf{hb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1})$. *This implies that* $\mathsf{pww} = (\mathsf{lhb}_{\mathbb{WW}} \cup \mathsf{cf}[\mathsf{lhb}])^+ \subseteq \mathsf{st} = ((\mathsf{hb}_{\mathbb{WR}} \circ \mathsf{wr}^{-1}) \cup \mathsf{hb}_{\mathbb{WW}})^+$.

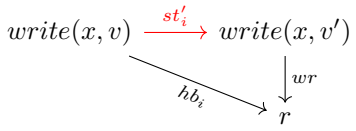


Figure 4.9 – Partial store order st'_i used to define wSC consistency model.

Finally, we can deduce that $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{pww} \cup \mathsf{rw}[\mathsf{pww}])^+ \subseteq \mathsf{hb} = (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{st} \cup \mathsf{rw}[\mathsf{st}])^+$. □

The reverse of this lemma does not hold. For example,

Example 30 *Figure 4.8b presents a* history *that satisfies CCM but not wSC. A possible partial store order for CCM is to consider that the writes of each thread are not visible to the other thread. This* history *does not satisfy wSC (See Example 29).*

Then,

Lemma 10 *wSC is strictly stronger than CCM.*

We prove now that wSC is weaker than SC. For that, we need to consider the subrelations of st and hb obtained by iterative least fix-point calculation. Let $\mathsf{st} = \bigcup_i st_i$ and $\mathsf{hb} = \bigcup_i hb_i$ where $st_i = (\mathsf{hb_{i\mathbb{WW}}} \cup st'_i)^+$ and st'_i (Fig.4.9) is defined by:

$$(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in st'_i \text{ iff } (\mathsf{write}(x,v), \mathsf{read}(x,v')) \in hb_i \text{ and } (\mathsf{write}(x,v'), \mathsf{read}(x,v')) \in \mathsf{wr}$$

where, for every $i \geq 0$, hb_i is defined by:

$$\begin{aligned} hb_0 &= (\mathsf{po} \cup \mathsf{wr})^+ \\ hb_{i+1} &= (hb_i \cup st_i \cup \mathsf{rw}[st_i])^+ \end{aligned}$$

We now show that the partial store order st_i is included in any store order ww witnessing for SC satisfaction.

Lemma 11 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ *be a* history *and* ww *be a total store order such that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ *is acyclic. Then,* $st_i \subseteq \mathsf{ww}$ *and* $hb_i \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$.

Proof 10 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ *be a* history *satisfying SC i.e., there exists a* store order ww *such that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ *is acyclic. We show that* $hb_i \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *and* $st_i \subseteq \mathsf{ww}$ *for all* ww *such that* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ *is acyclic. The proof is by induction on the index* i *of* hb_i *and* st_i.

Base-Case (i=0). *We have* $hb_0 = (\mathsf{po} \cup \mathsf{wr})^+$ *is included in* $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$. *Since* $hb_0 \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$, *if* $(\mathit{write}(x, v), \mathit{read}(x, v')) \in hb_0$ *and there exists a* $\mathit{read}(x, v')$ *such that* $(\mathit{write}(x, v'), \mathit{read}(x, v')) \in \mathsf{wr}$, *then* $(\mathit{write}(x, v), \mathit{write}(x, v')) \in \mathsf{ww}$. *Otherwise, assuming by contradiction that* $(\mathit{write}(x, v'), \mathit{write}(x, v)) \in \mathsf{ww}$, *we get* $(\mathit{read}(x, v'), \mathit{write}(x, v)) \in \mathsf{rw}$. *Since* $\mathit{write}(x, v), \mathit{read}(x, v')) \in hb_0 \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$, *this implies that there is a cycle in* $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *which is a contradiction. So, we have* $(\mathit{write}(x, v), \mathit{write}(x, v')) \in \mathsf{ww}$. *Thus,* st'_0 *is included in* ww *and hence* $st_0 = (\mathsf{hb}_{0\mathbb{WW}} \cup st'_0)^+$ *is also included in* ww *(since* $(\mathsf{hb}_{0\mathbb{WW}} \subseteq \mathsf{ww}$ *otherwise it leads to a contradiction since* $hb_0 \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *and* $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *is acyclic).*

Induction Step. *Assume that for all* ww, $hb_i \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *and* $st_i \subseteq \mathsf{ww}$. *Now, let's show that this holds for* $i + 1$ *as well. By induction hypothesis,* $st_i \subseteq \mathsf{ww}$, *so using the definition of* $\mathsf{rw}[st_i]$ *we have* $\mathsf{rw}[st_i] \subseteq \mathsf{rw}$. *Then,* $hb_{i+1} = (hb_i \cup st_i \cup \mathsf{rw}[st_i])^+ \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$. *Now, let's show that* $st'_{i+1} \subseteq \mathsf{ww}$. *If* $(\mathit{write}(x, v), \mathit{read}(x, v')) \in hb_i$ *and* $(\mathit{write}(x, v'), \mathit{read}(x, v')) \in \mathsf{wr}$, *then* $(\mathit{write}(x, v), \mathit{write}(x, v')) \in \mathsf{ww}$. *Otherwise, using the same argument in the base case, we get that* $(\mathit{read}(x, v'), \mathit{write}(x, v)) \in \mathsf{rw}$ *and a contradiction of the fact that* $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *is acyclic. Hence, if* $(\mathit{write}(x, v), \mathit{write}(x, v')) \in st'_{i+1}$ *then* $(\mathit{write}(x, v), \mathit{write}(x, v')) \in \mathsf{ww}$ *and so* $st'_{i+1} \subseteq \mathsf{ww}$. *Furthermore, we have* $\mathsf{hb}_{i+1\mathbb{WW}} \subseteq \mathsf{ww}$ *since* $hb_{i+1} \subseteq (po \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *(otherwise it leads to a contradiction of the fact that* $(po \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *is acyclic). Since* $st_{i+1} = (\mathsf{hb}_{i+1\mathbb{WW}} \cup st'_{i+1})^+$, $st'_{i+1} \subseteq \mathsf{ww}$ *and* $\mathsf{hb}_{i+1\mathbb{WW}} \subseteq \mathsf{ww}$, *we get that* $st_{i+1} \subseteq \mathsf{ww}$ *(since* ww *is a total store order).* □

As an immediate corollary of Lemma 11, we get:

Lemma 12 *If a* history *satisfies SC, then it satisfies wSC.*

Proof 11 *The proof is by contradiction. Assume that a history* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ *satisfies SC and it does not satisfy wSC. Since* h *satisfies SC, there exists a total*

store order ww *such that* po∪wr∪ww∪rw *is acyclic. Since h does not satisfy wSC, this means that hb is cyclic. Since* $\mathsf{hb} = \bigcup_i hb_i$ *and* $hb_i \subseteq (\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *(from Lemma 11), we can deduce that* $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ *is also cyclic and this constitutes a contradiction.* □

The reverse of the above lemma doesn't hold. For instance,

Example 31 *Figure 4.8a shows a* history *which satisfies wSC (the reason has been shown above in example 28) but it is not SC. Since there is no valid store order for the writes* $\mathsf{write}(z,1)$ *and* $\mathsf{write}(z,2)$*, this* history *does not satisfy SC. In fact, since ww is a total order, let's try the two possible cases:*

- *If* $\mathsf{write}(z,1)$ *happens-before* $\mathsf{write}(z,2)$ *in the total order* ww*, then* $\mathsf{write}(y,1)$ *precedes* $\mathsf{write}(y,2)$ *in* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww}$*. Since,* $\mathsf{read}(y,1)$ *reads its value from* $\mathsf{write}(y,1)$*, we get a* rw *relation between* $\mathsf{read}(y,1)$ *and* $\mathsf{write}(y,2)$*. In the same way,* $\mathsf{write}(x,1)$ *precedes* $\mathsf{write}(x,2)$ *in* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww}$*, then, we get a* rw *between* $\mathsf{read}(x,1)$ *(which takes its value from* $\mathsf{write}(x,1)$*) and* $\mathsf{write}(x,3)$*. Thus, we get a cycle in* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$*.*
- *If* $\mathsf{write}(z,2)$ *precedes* $\mathsf{write}(z,1)$ *in* ww*, then* $\mathsf{write}(t,1)$ *precedes* $\mathsf{write}(t,2)$ *in* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww}$*. Since,* $\mathsf{read}(t,1)$ *takes its value from* $\mathsf{write}(t,1)$*,* $\mathsf{read}(t,1)$ *should precede* $\mathsf{write}(t,2)$ *in read-write relation* rw*. Similarly,* $\mathsf{read}(s,1)$ *should precede* $\mathsf{write}(s,2)$ *in* rw*. Then, we get a cycle in* $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$*.*

Both cases are not possible (i.e., lead to a cycle in $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$*), then the history is not allowed by SC.*

Thus,

Lemma 13 *SC is strictly stronger than wSC.*

The Figure 4.10 completes the Figure 4.3 by the relationships between the consistency models studied in this section. In this chapter, we have proposed CCM that is strictly stronger than all known causal consistency models (CC, CCv and CM) and strictly weaker than SC. We have also introduced wSC that is strictly stronger than CCM (and explicitly strictly stronger than CC, CCv and CM) and strictly weaker than SC.

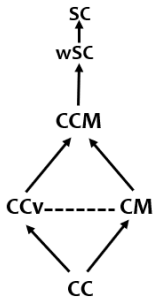


Figure 4.10 – Relationships between consistency models: CC, CCv, CM, CCM, wSC and SC

4.2.2 The Sequential Consistency Kernel

Given a history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ that satisfies SC, we define the Sequential Consistency Kernel (SC-Ker for short) of h as the intersection of all store order orders allowing to establish the SCness of h. We know already, from the previous section (Lemma 11), that the store order st, computed by the wSC saturation procedure, is included in any total store order ww such that $\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$ is acyclic. This means that the computed st is always a subset of SC-Ker. Then, the question is whether the computed store order st is equal to SC-Ker or not.

In the following, we show that the saturation procedure of wSC may in some cases not be able to compute the SC-Ker (but rather a strict subset of it). To see why, consider the history given in Figure 4.11.

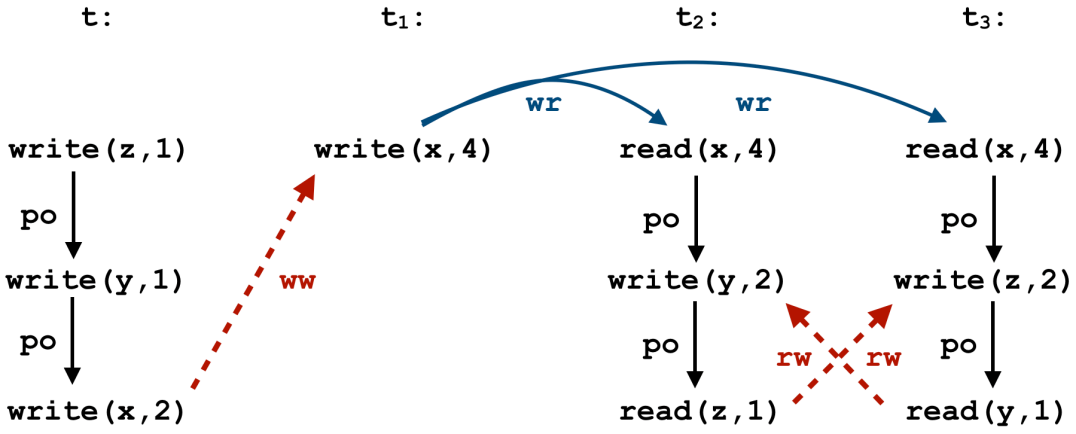


Figure 4.11 – SC-Kernel counter example

The wSC rules do not generate any st relation and therefore the saturation

procedure of wSC returns that the store order st is empty while the happens-before relation hb is equal to $(\mathsf{po} \cup \mathsf{wr})^+$. However, any total store order ww that allows to show the SCness of this history should order $\mathsf{write}(x, 4)$ before $\mathsf{write}(x, 2)$ (and hence the pair $(\mathsf{write}(x, 4), \mathsf{write}(x, 2))$ is in the SC-Ker). We prove that $(\mathsf{write}(x, 4), \mathsf{write}(x, 2))$ belongs to the SC-Ker by contradiction. Assume that $(\mathsf{write}(x, 4), \mathsf{write}(x, 2))$ is not in SC-Ker. Then, there is a total store order ww such that (1) $(\mathsf{write}(x, 2), \mathsf{write}(x, 4))$ is in ww (represented in Figure 4.11 by a dashed arrow) and (2) $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ is acyclic (since the history h is SC). However, if $(\mathsf{write}(x, 2), \mathsf{write}(x, 4))$ is in ww then the relation $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ is not acyclic (as shown in Figure 4.11 by the dashed arrows) and which is a contradiction.

One way to overcome this problem is to include such a pattern in the definition of the total order st'_i used in the saturation procedure. Thus, the new definition of st'_i is updated as follows: $(\mathsf{write}(x, v'), \mathsf{write}(x, v)) \in st'_i$ if and only if one of the following cases holds:

- $(\mathsf{write}(x, v'), \mathsf{read}(x, v)) \in hb_i$ and $(\mathsf{write}(x, v), \mathsf{read}(x, v)) \in \mathsf{wr}$, or
- $(\mathsf{write}(z, v_z), \mathsf{write}(x, v))$, $(\mathsf{write}(y, v_y), \mathsf{write}(x, v))$, $(\mathsf{write}(x, v'), \mathsf{write}(y, v'_y))$, $(\mathsf{write}(y, v'_y), \mathsf{read}(z, v_z))$, $(\mathsf{write}(x, v'), \mathsf{write}(z, v'_z))$, $(\mathsf{write}(z, v'_z), \mathsf{read}(y, v_y))$ are in hb_i and $(\mathsf{write}(z, v_z), \mathsf{read}(z, v_z))$, $(\mathsf{write}(y, v_y), \mathsf{read}(y, v_y))$ are in wr.

Observe that the pattern added to st'_i contains six write operations. Unfortunately, this pattern is not enough to allow us to capture the SC-Ker. In fact, we can construct a family of counter-examples (see Figure 4.12) such that in order to capture all of them, we need to add to the relation st'_i patterns involving a strictly increasing number of writes (which is not feasible in practice).

One way to address the problem raised by the family of counter-examples given in Figure 4.12 is to guess for a given pair of writes $\mathsf{write}(x, v)$ and $\mathsf{write}(x, v')$ that are not related by the computed store relation st (i.e., $(\mathsf{write}(x, v), \mathsf{write}(x, v'))$ and $(\mathsf{write}(x, v'), \mathsf{write}(x, v))$ are not in st) one possible order and check if it can make the history h infeasible under SC and if it is the case we add the other order to st. For instance, in the history given in Figure 4.11, one would guess that the $(\mathsf{write}(x, 2), \mathsf{write}(x, 4))$ is in st. This guess makes the history infeasible under SC due to the existence of a cycle in $(\mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw})^+$ and hence $(\mathsf{write}(x, 4), \mathsf{write}(x, 2))$ is added to st. Observe that this still results

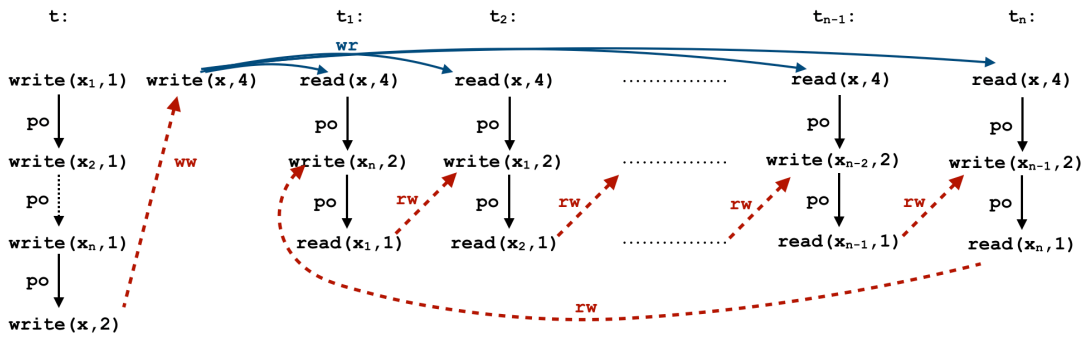


Figure 4.12 – SC-Kernel counter examples with cycles involving an arbitrary number of writes

in a saturation procedure which works in polynomial time since, at any time, we guess the order of at most two unrelated writes.

So the question is whether this extended saturation procedure calculates the SC-Ker. Alas, this is not true. Consider the history given in Figure 4.13. The previous saturation procedure of wSC (augmented with the guessing of the order of one pair of writes) results in an empty store order st. However, this history satisfies SC and $(\mathsf{write}(x,1), \mathsf{write}(x,2))$ and $(\mathsf{write}(t,2), \mathsf{write}(t,1))$ are in SC-Ker. In fact, ordering $\mathsf{write}(x,2)$ before $\mathsf{write}(x,1)$ and $\mathsf{write}(t,2)$ before $\mathsf{write}(t,1)$ creates a happens-before cycle in the top-left block of Figure 4.13 (in similar manner to the example given in Figure 4.11). While ordering $\mathsf{write}(x,2)$ before $\mathsf{write}(x,1)$ and $\mathsf{write}(t,1)$ before $\mathsf{write}(t,2)$ creates a happens-before cycle in the top-right block of Figure 4.13. Finally, ordering $\mathsf{write}(x,1)$ before $\mathsf{write}(x,2)$ and $\mathsf{write}(t,1)$ before $\mathsf{write}(t,2)$ creates a happens-before cycle in the top-middle block of Figure 4.13.

This shows the necessity of augmenting the saturation procedure with the enumeration of the order between two pairs of writes in order to compute the SC-Ker. Even worst, we can easily extend the history given in Figure 4.13 in order to force the enumeration of the order between several pairs of writes in order to be able to compute the SC-Ker. The main idea is to add a number of blocks (in similar manner to the examples given in Figure 4.11 and Figure 4.12) to forbid all order combinations between certain pairs of write except one.

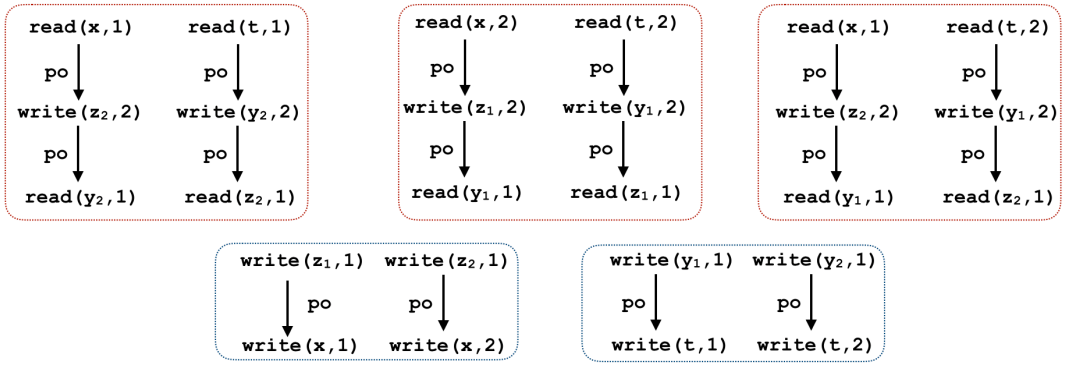


Figure 4.13 – SC-Kernel counter requiring the enumeration of the possible order between two pairs of writes

4.2.3 Algorithms for checking SC conformance

We define in this section algorithms for SC checking that exploit the partial store order st computed by the wSC saturation. Following the approach of gradual consistency checking (GCC) we have seen in the first part of this chapter, we start by checking that the given history is wSC. If not, then we conclude that it is not SC neither (by Lemma 12). If yes, we exploit st in order to enhance the SC verification of the history. This verification amounts in finding a total store order extending st. To solve this problem we adopt two approaches, one is based on reducing the SC verification problem to SAT, and the second one is based on using the bounded-thread approach of [7, 18] implemented in the DBCOP tool. Both of these approaches are enhanced by the fact that they will receive the st constraints in order to reduce their search space. The two so obtained algorithms are called wSC+ENUM and wSC+DBCOP.

The algorithm wSC+ENUM uses an encoding of SC conformance of a given history (defined with its po and wr constraints) as the satisfaction of a boolean formula. The latter expresses the constraints on the relations involved in the definition of SC, including the fact that the *store oder* ww is a total order relation (so every pair of writes must be order in one direction or the other), and that the happen-before relation is transitive and acyclic. Moreover, the order constraints corresponding to the relation st computed for wSC are added to the formula. Hence,

Theorem 6 *Algorithm 4 returns true iff the input history h satisfies SC.*

Algorithm 4: Checking SC conformance: wSC+ENUM algorithm.

Input: A history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$
Output: *true* iff h satisfies SC

```
1 if po ∪ wr ∪ st ∪ rw[st] is cyclic then
2     return false;
3 foreach ww ⊃ st do
4     if po ∪ wr ∪ ww ∪ rw[ww] is acyclic then
5         return true;
6 return false;
```

Algorithm 5: Checking SC conformance: wSC+DBCOP algorithm.

Input: A history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$
Output: *true* iff h satisfies SC

```
1 if po ∪ wr ∪ st ∪ rw[st] is cyclic then
2     return false;
3 if DBCOP(po, wr, st) then
4     return true;
5 return false;
```

The algorithm wSC+DBCOP is based on the algorithm implemented in DBCOP [18]. Given a history (again defined by its po and wr relations), DBCOP searches for an interleaving of all the operations of the history that respects the constraints imposed by SC. Then, wSC+DBCOP is an adaptation of DBCOP that exploits st in addition to po and wr as fixed constraints during its search. Hence,

Theorem 7 *Algorithm 5 returns true iff the input history h satisfies SC.*

For our experiments in next section, we compare wSC+ENUM and wSC+DBCOP to each other, to DBCOP, and also to CCM+ENUM which is the analogous of wSC+ENUM using CCM saturation instead of wSC saturation. CCM+ENUM is the algorithm defined in the first SC checking approach (Section 4.1).

4.2.4 Complexity

Notice that at each step of the calculation of hb and st, at least one pair of operations is added to one of these two relations and that the number of such

pairs is polynomially bounded (in the size of the computation), there can be at most n^2 edges where n is the number of operations in the computation. As we have seen in the hb and st saturation rules, it involves 3 operations ($\mathcal{O}(n^3)$). Thus, the acyclicity of hb can be decided in polynomial time. Then,

Theorem 8 *Checking whether a* history *h satisfies wSC is polynomial time* ($\mathcal{O}(n^5)$) *in the size of the* history.

4.2.5 Experimental Evaluation

We show in this section an evaluation of the efficiency of our approach and its scalability. We first report on the efficiency of the wSC saturation in computing the SC-kernel. Then, we present an evaluation of the approach in checking SC conformance by taking into account two dimensions: the number of operations and the number of threads. The evaluation examines the case of valid histories (that satisfy SC), the case of histories that violate SC, and the case where both types of histories are considered. Alike the first section, experiments are done by considering histories that are generated by running random clients (ruby random tester [6]) on realistic cache coherence protocols within the Gem5 simulator [17] in system emulation mode. We used 4 cache coherence protocols included in Gem5: MI, MEOSI Hammer, MESI Two Level, and MEOSI AMD Base. We use another implementation of CCM+ENUM which is more efficient compared to the Datalog based approach we have considered in the Section 4.1.

The Figure 4.14 shows the general schema of the wSC-based testing procedure we used in our experiments to check SC.

Capacity of ordering the set of writes

We evaluate the capacity of CCM and wSC in computing store order constraints that must be part of any store order witnessing SCness. In fact, we know that their saturation procedures compute subsets of the SC-kernel of SC histories. The questions we address is what is the computed proportion of this set, and what is the proportion of the set of pairs of writes in the execution that are not ordered by saturation (including those that are outside the SC-kernel when the history is SC).

We found that wSC computes the SC-kernel in 74.24% of the tested execu-

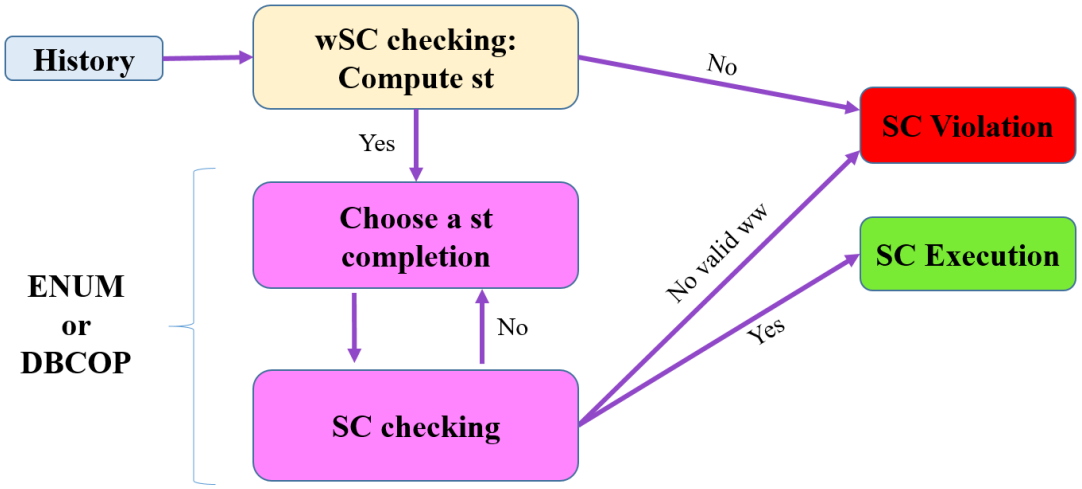


Figure 4.14 – The general schema of the SC checking procedure using wSC

tions, and that for the rest of the executions, it computes in average 99.97% of their kernel. As for CCM, we found that it computes the SC-kernel only in 0.7% of the same set of executions.

We also found that wSC saturation puts 98.51% of the pairs of writes of an execution in average, and that CCM order in average 97,89% of the pairs of writes.

This is interesting since in terms of coverage of the sets of pairs of write, CCM is not far from wSC, however, only for very few execution it can cover fully its SC-kernel.

SC conformance checking: Valid histories

We consider in this section the case of executions that satisfy SC. The experiments are made by varying the number of operations and the number of threads. For each value of number of operations, of threads, respectively, we have tested 200 histories and computed the running time average.

Figure 4.15 reports the running time of the 4 algorithms wSC+ENUM, CCM+ENUM, DPCOP, and wSC+DBCOP while increasing the number of operations from 200 to 800, in increments of 100 with a fixed number of 6 threads. It shows that for a fixed, relatively small number of threads, DBCOP has the best performances, while wSC+ENUM good performances and is clearly superior than CCM+ENUM. This is due partly to the difference in the coverage of

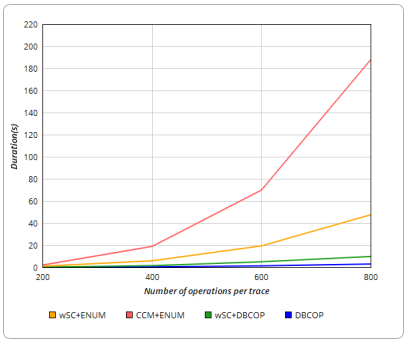


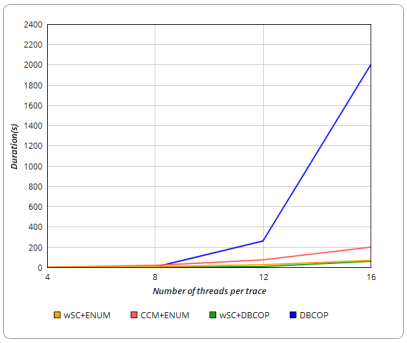
Figure 4.15 – Checking SC for valid histories while varying the number of operations.

store order constraints, but only because the difference between the two coverage percentages is not big in average (98.51% vs 97,89%). So, here the complexity of the saturation technique plays also in important role: for CCM, the saturation schema requires computing local happen-before relation for each operation, which is very expensive compared with the much simpler saturation schema in wSC.

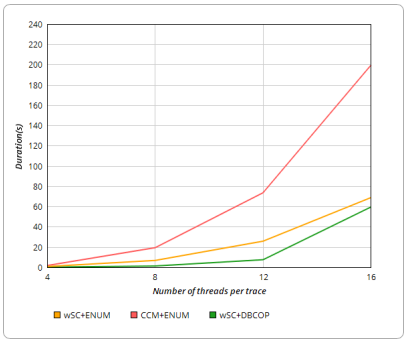
Figure 4.16 reports the running time while increasing the number of threads from 4 to 16, in increments of 4. We have considered 50 operations per thread. Notice that increasing the number of threads increases the global number of operations too. Figure 4.16(a) shows that the performances of DBCOP degrade quickly beyond 8 threads, while the other algorithms exploiting saturation are more scalable, wSC+ENUM being better than CCM+ENUM, wSC+DBCOP achieving the best performances. Figure 4.16(b) is a zoom of Figure 4.16(a) for a smaller time scale in order to examine more closely the separation between CCM+ENUM, wSC+ENUM, and wSC+DBCOP. It can be seen that the combination of wSC saturation with DBCOP leads to an efficient procedure that takes advantage from the DBCOP strategy for small number of threads, and exploits wSC saturation to stay scalable when both the number of threads and operations increase.

SC conformance checking: Valid and invalid histories

We now consider a set of histories containing 50% of violations. The violations are generated by randomly changing the write-read relation: for some number of reads chosen randomly, we modify their return value by choosing the one written by a write operation taken randomly in the execution within some

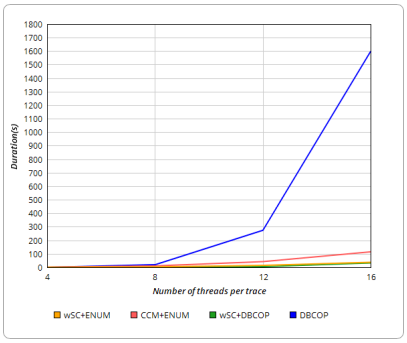


(a) Comparing all approaches.

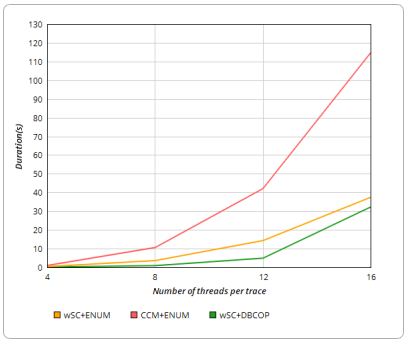


(b) Comparison of wSC+ENUM, CCM+ENUM and wSC+DBCOP.

Figure 4.16 – Checking SC for valid histories while varying the number of threads.



(a) Comparing all approaches.



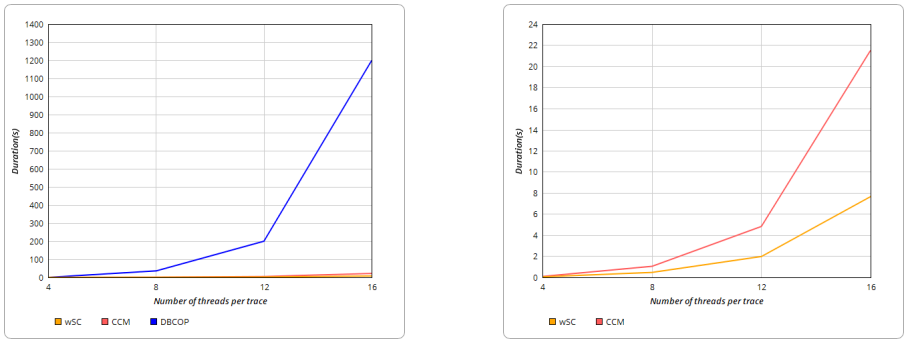
(b) Comparison of wSC+ENUM, CCM+ENUM and wSC+DBCOP.

Figure 4.17 – Checking SC for a set of 50% of valid and 50% of invalid histories.

bounded distance from the considered read. As in the previous paragraph, we consider histories with 4 to 16 threads and we test 200 histories for each number of threads. The results shown in Figure 4.17b and Figure 4.17a show that the considered algorithms behave and compare in the case with mixed types of histories similarly to the case with only valid histories.

SC conformance checking: Invalid histories

We consider now the case of only violations. We consider histories with 4 to 16 threads and 50 operations per thread. For each number of threads, we consider 100 histories and compute the average running time. Since all found



(a) Comparison of wSC, CCM and DBCOP.

(b) Comparison of wSC and CCM.

Figure 4.18 – Checking SC for invalid histories.

violations are already wSC violations, we only compare the saturation steps wSC and CCM with DBCOP. Figure 4.18b shows that clearly wSC is more efficient than CCM. In addition, wSC captures more SC violations: 1,25% of the violations are not captured by CCM. Figure 4.18 shows that wSC has better performance, by factors of 70 times (in the 8 threads case) and higher, compared to DBCOP (the latter crashes when the number of threads is large) while wSC is very efficient, it terminates in less than 8 seconds for all the tested histories. This shows the superiority of saturation in detecting quickly consistency violations, and it scales very well when increasing the number of threads (and therefore the total number of operations as well).

4.3 Conclusion

We have proposed two approaches for checking SC conformance. The idea over these approaches is to avoid an explicit enumeration of the exponential number of possible total orders between writes in order to solve these problem. Our approach is to define weaker criteria that are as strong as possible but still polynomial time checkable. Morally, the approach consists in being able to capture an “as large as possible” partial order on writes that can be computed in polynomial time (using a least fixpoint calculation), and which is a subset of any total order witnessing SC conformance. Then, the first idea was to exploit the existing causal consistency models and combine them in order to come out with a strong causal consistency variant (CCM) which is polynomially checkable and still

weaker than SC. The experiments confirmed that this approach allows to reduce significantly the number of pairs of writes for which an order should be found in order to establish SC conformance in an enumerative way. However, we have shown that this approach can be pushed more and we have introduced another consistency model (wSC) which is stronger than CCM, polynomially checkable and weaker than SC. Our experimental results showed that in practice (1) this allows to catch very quickly almost all SC-violations, and (2) our method allows to compute almost always the whole SC-kernel (around 99.9% of it), and leaves only a very small number of store order constraints to be found in order to check SC-ness. We considered two ways for finding the remaining constraints: either using SAT-solving, or using the search procedure of DBCOP. The latter option, exploiting saturation to enhance DBCOP, is the best one experimentally, leading to a performant and scalable algorithm. An interesting question is whether this approach can be generalized in order to cover other consistency models for which the conformance verification problem is NP-hard.

We address this problem in the next chapter and propose similar approaches for checking Total Store Order (TSO) conformance.

CHAPTER 5

TOTAL STORE ORDERING VERIFICATION

In this chapter, we address the problem of verifying that an execution is conform to the TSO consistency model. This problem is known to be NP-complete [41, 38] as the SC checking problem. This similarity is due to the fact that in order to justify that the execution is consistent, one has to find a total order between the writes which explains the read operations happening along the computation. It can be proved that one cannot avoid enumerating all the possible total orders between writes, in the worst case.

For the case of TSO, we proceed in the same way as for SC, but we consider different intermediary polynomial time checkable criteria. This is due to the fact that some constraints need to be relaxed under TSO in order to take into account the program order relaxations of TSO (ppo and po-loc), that allow reads to overtake writes. As we have seen in the first chapter, a *history* $\langle O, \mathsf{po}, \mathsf{wr} \rangle$ satisfies TSO if there exists a *store order* ww such that $\mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ and $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ are both acyclic.

Our first approach is based on a weakening of CCM called weak CCM (wCCM), that is weaker than TSO and polynomial time checkable. Then, given a history, if it is a violation of wCCM then the history is not conform to TSO as well (TSO is stronger than wCCM). Otherwise, if the history satisfies wCCM, we try to find an extension of the computed partial store order (the order between writes imposed by wpww) that can witness for TSO satisfaction. The soundness of this extension is implied by the fact the constraints imposed by wCCM on the writes

order are included in any total store order witnessing for TSO conformance.

Our experiments show a significant improvement of TSO checking performance when we use wCCM as an upper-approximation of TSO w.r.t. a standard enumeration using a SAT solver. However, an interesting question is whether wCCM is the strongest consistency model we can use in this approach.

Similarly to SC, our second approach extends this idea by proposing a new consistency model, called weak TSO (wTSO), that is stronger than wCCM, weaker than TSO and the most important point is that it is checkable in polynomial time.

Our second approach relies on the mentioned (new) consistency model called *weak Total Storing Order* (wTSO, for short) that is based on a simple *saturation rule* for imposing constraints on *store order*. The idea is to apply the saturation rule to a pair of writes in order to avoid a cycle in the happen-before relation involving a conflict including one of the writes. The proposed saturation rule in wTSO is more simpler compared to the one used for wCCM. Furthermore, we prove that wTSO is stronger than wCCM. While the first part of this chapter is published in [69], the second part is an extension of the work published in [65].

This chapter is structured in two sections. The first section (Section 5.1) is dedicated to the TSO checking approach which is based on using wCCM as an upper-approximation of TSO. We first define the wCCM consistency model and its related relations. Then, we show that wCCM is weaker than TSO. Afterwards, we present our algorithm for checking TSO conformance and evaluate it using the same real executions used in the SC case. The second section (Section5.2) presents wTSO based approach for checking TSO. First, we define the wTSO consistency model. Then, we prove that wTSO is stronger than wCCM and weaker TSO, and discuss the wTSO checking complexity. Finally, we present our algorithm for checking TSO using wTSO based approximation and evaluate it on real executions.

5.1 Approach 1: wCCM-based TSO verification

We define a polynomial time checkable criterion, called weak convergent causal memory (wCCM, for short), based on a (different) variation of causal consistency that is suitable for the case of TSO. This allows to reduce the number of pairs of

writes for which an order must be guessed in order to establish the conformance to TSO.

The case of TSO requires the definition of a new intermediary consistency model because CCM is based on a causality order that includes the program order po which is relaxed in the context of TSO, compared to the SC model. Indeed, CCM is *not* weaker than TSO as shown by the history in Figure 5.1a (note that this does not imply that other variations of causal consistency, CC and CCv, are also not weaker than TSO). This history satisfies TSO because, based on its operational model, the operation $\mathsf{write}(x, 2)$ of thread t_1 can be delayed (pending in the store buffer of t_1) until the end of the execution. Therefore, after executing $\mathsf{read}(z, 0)$, all the writes of thread t_0 are committed to the main memory so that thread t_1 can read 1 from y and 2 from x (it is obliged to read the value of x from its own store buffer).

This history is not admitted by CCM because it is not admitted by the weaker causal consistency variation CM. Figure 5.6b presents a history admitted by CCM but not by TSO. Indeed, under TSO, both t_2 and t_3 should observe the writes performed by t_0 and t_1, respectively, on variable x and variable y in the same order. However, it does not, because t_2 sees the write on variable x before the write on variable y (since it reads 0 from y) and t_3 sees the write on y before the write on x (since it reads 0 from x). This history is admitted by CCM since the two writes are causally independent and they concern different variables.

We mention that TSO and CM are also incomparable. As we have seen in the chapter 2, the history in Figure 5.1b is allowed by CM, but not by TSO. The history in Figure 5.1a is admitted by TSO, but not by CM. Then, CM and CCM cannot be used to approximate TSO.

Next, we define the weakening of CCM, called *weak convergent causal memory* (wCCM), which is also weaker than TSO. The wCCM model is actually based on causality relations induced by the relaxed program orders ppo and po-loc instead of po, and the external write-read relation instead of the full write-read relation.

Section 5.2.1 introduces wCCM while Section 5.1.2 is dedicated to our algorithm for verifying TSO based on wCCM. Section 5.1.4 presents the experimental results.

t_1:	t_2:
$\mathsf{write}(z,1)$	$\mathsf{write}(x,2)$
$\mathsf{write}(x,1)$	$\mathsf{read}(z,0)$
$\mathsf{write}(y,1)$	$\mathsf{read}(y,1)$
	$\mathsf{read}(x,2)$

(a) TSO but not CM nor CCM

t_0:	t_1:
$\mathsf{write}(x,1)$	$\mathsf{write}(x,2)$
$\mathsf{read}(x,2)$	$\mathsf{read}(x,1)$

(b) CM but not CCv nor wCCM nor TSO

t_0:	t_1:	t_2:	t_3:
$\mathsf{write}(x,1)$	$\mathsf{write}(y,1)$	$\mathsf{read}(x,1)$	$\mathsf{read}(y,1)$
		$\mathsf{read}(y,0)$	$\mathsf{read}(x,0)$

(c) wCCM and CCM but not TSO.

Figure 5.1 – Comparison of CM, wCCM, CCM and TSO consistency models.

5.1.1 Weak Convergent Causal Memory

First, we define two causality relations relative to the partial program orders in the definition of TSO and the external write-read relation: For $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, let $\mathsf{co}^\pi = (\pi \cup \mathsf{wr}_e)^+$. We also consider a notion of conflict that is defined in terms of the external write-read relation as follows: For a given relation R, let $\mathsf{cf}_e[R] = R_{\mathbb{WR}} \circ \mathsf{wr}_e^{-1}$ (Figure 5.2a) .

Then, given a history $\langle O, \mathsf{po}, \mathsf{wr}\rangle$, we define for each operation o two happens-before relations $\mathsf{lhb}_o^{\mathsf{ppo}}$ and $\mathsf{lhb}_o^{\mathsf{po\text{-}loc}}$. The definition of these relations is similar to the one of lhb_o (from causal memory (CM)), the differences being that po is replaced by ppo and $\mathsf{po\text{-}loc}$ respectively, co is replaced by $\mathsf{co}^{\mathsf{ppo}}$ and $\mathsf{co}^{\mathsf{po\text{-}loc}}$ respectively, and wr is replaced by wr_e. Therefore, for $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, lhb_o^π is the smallest transitive relation such that:

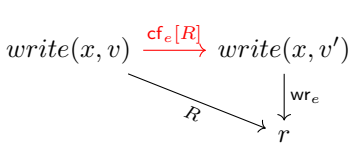
1. $(o_1, o_2) \in \mathsf{lhb}_o^\pi$ if $(o_1, o_2) \in \mathsf{co}^\pi$, $(o_1, o) \in \mathsf{co}^\pi$, and $(o_2, o) \in (\mathsf{co}^\pi)^*$, and
2. $(\mathsf{write}(x,v), \mathsf{write}(x,v')) \in \mathsf{lhb}_o^\pi$ if $(\mathsf{write}(x,v), \mathsf{read}(x,v')) \in \mathsf{lhb}_o^\pi$, and $(\mathsf{write}(x,v'), \mathsf{read}(x,v')) \in \mathsf{wr}$ and $(\mathsf{read}(x,v'), o) \in \pi^*$, for some $\mathsf{read}(x,v')$ (Figure 5.2c).

Let $\mathsf{lhb}^\pi = (\bigcup_{o\in O} \mathsf{lhb}_o^\pi)^+$, for $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, and let $\mathsf{wlhb} = (\mathsf{lhb}_o^{\mathsf{ppo}} \cup \mathsf{lhb}_o^{\mathsf{po\text{-}loc}})^+$. Then, the weak partial store order is defined as follows:

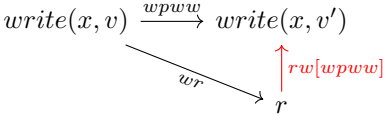
$$\mathsf{wpww} = (\mathsf{wlhb}_{\mathbb{WW}} \cup \mathsf{cf}_e[\mathsf{lhb}^{\mathsf{po\text{-}loc}}] \cup \mathsf{cf}_e[\mathsf{lhb}^{\mathsf{ppo}}])^+$$

Then,

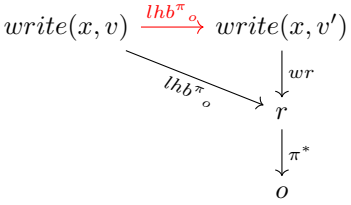
Definition 17 *A* history $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ *satisfies weak Convergent Causal Memory*



(a) Conflict order cf[R]



(b) Read-write rw[wpww]



(c) Local happen-before lhb^{π}_{o}

Figure 5.2 – Definitions of relations used to define wCCM consistency model.

(wCCM) if both relations:

$$\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{wpww} \cup \mathsf{rw}[\mathsf{wpww}] \text{ and } \mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{wpww} \cup \mathsf{rw}[\mathsf{wpww}]$$

are acyclic.

As we have seen, the external write-read relation wr_e is a restriction of the write-read relation to pairs that are not in the same thread, i.e., $\mathsf{wr}_e = \mathsf{wr} \cap \{(o, o') \mid (o, o') \notin \mathsf{po} \text{ and } (o', o) \notin \mathsf{po}\}$. The read-write relation $\mathsf{rw}[wpww]$ induced by wpww (Figure 5.2b) is defined by $\mathsf{rw}[wpww] = \mathsf{wr}^{-1} \circ wpww$.

To exemplify, consider the following examples.

Example 32 *The history in Figure 5.1c is allowed by wCCM. The reason is that the two writes (*$\mathsf{write}(x, 1)$ *and* $\mathsf{write}(y, 1)$*) are not causally related and they are written in different variables* x *and* y*.*

Example 33 *The history in Figure 5.1b does not satisfy wCCM. Since* $\mathsf{write}(x, 1)$ *precedes* $\mathsf{read}(x, 2)$ *in* $\mathsf{po\text{-}loc}$ *and* $\mathsf{read}(x, 2)$ *in* t_0 *takes its value from* $\mathsf{write}(x, 2)$ *in the* t_1 *(i.e.,* $(\mathsf{write}(x, 2), \mathsf{write}(x, 2)) \in \mathsf{wr}_e$*),* $\mathsf{write}(x, 1)$ *should precede* $\mathsf{write}(x, 2)$ *in* cf_e *relation. Similarly, since* $\mathsf{write}(x, 2)$ *precedes* $\mathsf{read}(x, 1)$ *in* $\mathsf{po\text{-}loc}$ *and* $\mathsf{read}(x, 1)$ *in* t_1 *takes its value from* $\mathsf{write}(x, 1)$ *in the* t_0 *(i.e.,* $(\mathsf{write}(x, 1), \mathsf{write}(x, 1)) \in \mathsf{wr}_e$*),* $\mathsf{write}(x, 2)$ *should precede* $\mathsf{write}(x, 1)$ *in* cf_e *relation. Then, we get a cycle in* cf_e *and the history is not wCCM.*

We prove that TSO is stronger than wCCM.

Lemma 14 *If a history satisfies TSO, then it satisfies wCCM.*

Proof 12 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ *be a history satisfying TSO. Then, there exists a store order* ww *such that* $\mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ *and* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ *are both acyclic. The fact that*

$$\mathsf{lhb}^{\mathsf{po\text{-}loc}} \subseteq (\mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+ \text{ and } \mathsf{lhb}^{\mathsf{ppo}} \subseteq (\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$$

can be proved by structural induction like in the case of SC (the step of the proof showing that $\mathsf{lhb} \subseteq \mathsf{po} \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}]$*). Then, since* ww *is a total order on writes on the same variable, we get that the projection of* wlhb *(the transitive closure of the union of* $\mathsf{lhb}^{\mathsf{po\text{-}loc}}$ *and* $\mathsf{lhb}^{\mathsf{ppo}}$*) on pairs of writes on the same variable is included in* ww*. Therefore,* $\mathsf{wlhb}_{\mathbb{WW}} \subseteq \mathsf{ww}$*. Then, since* $\mathsf{cf}_e[R^\pi] \subseteq R^\pi$ *for each* $R^\pi = (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *with* $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$ *and since each* $\mathsf{cf}_e[R^\pi]$ *relates only writes on the same variable, we get that each* $\mathsf{cf}_e[R^\pi]$ *is included in* ww*. This implies that* $\mathsf{wpww} \subseteq \mathsf{ww}$*.*

Finally, since $\mathsf{wpww} \subseteq \mathsf{ww}$*, we get that* $(\pi \cup \mathsf{wr} \cup \mathsf{wpww} \cup \mathsf{rw}[\mathsf{wpww}])^+ \subseteq (\pi \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}[\mathsf{ww}])^+$*, for each* $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$*. In each case, the acyclicity of the latter implies the acyclicity of the former. Therefore,* h *satisfies wCCM.* □

The reverse of the above lemma does not hold. Indeed, it can be easily seen that wCCM is weaker than CCM (since wpww is included in pww). The following example shows a history that satisfies CCM (then wCCM as well) but not TSO.

Example 34 *The history in Figure 5.6b satisfies CCM but not TSO (as explained before) then it satisfies wCCM but not TSO (wCCM is strictly weaker than CCM).*

Then,

Lemma 15 *TSO is strictly stronger than wCCM.*

We now compare wCCM to CM.

wCCM compared to CM: Consider the following examples,

Example 35 *The history in Figure 5.6b is allowed by wCCM (since it is allowed by TSO, as explained in the beginning of the section), but not by CM.*

Example 36 *Since CCM is stronger than CM, the history in Figure 5.1b satisfies CM but not wCCM (As we have seen above).*

Then,

Result 2 *wCCM and CM are incomparable.*

The relationships between the consistency models that we have seen above are summarized in Figure 5.3. Establishing the precise relation between CC/CCv and TSO is hard because of the fact that CC and CCv are defined using one acyclicity condition while TSO is based on two acyclicity conditions. We believe that CC and CCv are weaker than TSO, but we do not have a formal proof.

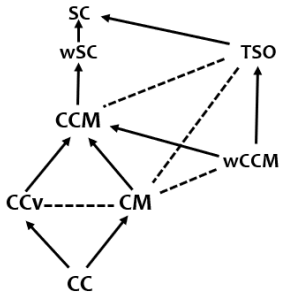


Figure 5.3 – Relationships between consistency models: CC, CCv, CM, CCM, wSC, wCCM, TSO and SC.

5.1.2 An Algorithm for Checking TSO conformance using wCCM

The wCCM-based algorithm for checking TSO conformance for a given history is presented in Algorithm 6. It starts by checking whether the history violates the weaker consistency model wCCM. If yes, it returns false. If not, it starts enumerating the orders between the writes that are not related by the weak partial store order wpww until it founds one that allows establishing TSO conformance. So, in this case it returns true. Otherwise, if no valid total order is found, it returns false. This enumeration can be done either using standard enumeration or using a SAT solver.

Theorem 9 *Algorithm 6 returns true iff the input history h satisfies TSO.*

Algorithm 6: Checking TSO conformance: wCCM+ENUM algorithm.

Input: A history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$
Output: *true* iff h satisfies TSO

```
1 if ppo ∪ wr_e ∪ wpww ∪ rw[wpww] or po-loc ∪ wr_e ∪ pww ∪ rw[wpww] is cyclic then
2     return false;
3 foreach ww ⊃ wpww do
4     if ppo ∪ wr_e ∪ ww ∪ rw[ww] and po-loc ∪ wr_e ∪ ww ∪ rw[ww] are acyclic then
5         return true;
6 return false;
```

5.1.3 Complexity

It can be seen that similarly to pww, the weak partial store order wpww can be computed in polynomial time (in the size of the input history). In fact, the lhb^{π} relations (for each $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$) can be computed in at most a quadratic number of iterations (using a least fix-point calculation for instance). the cf_e can be computed using a least fix-point calculation as well, and the acyclicity of:

$$\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{wpww} \cup \mathsf{rw}[\mathsf{wpww}] \text{ and } \mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{wpww} \cup \mathsf{rw}[\mathsf{wpww}]$$

can be decided in polynomial time. Thus,

Theorem 10 *Checking whether a history satisfies wCCM is polynomial time* ($\mathcal{O}(n^5)$) *in the size of the history.*

5.1.4 Experimental Evaluation

In order to evaluate the efficiency and scalability of the proposed approach, we have implemented the TSO algorithms we introduced above and use them in experiments to check the TSO conformance. We have investigated their running time, compared to a standard encoding of TSO model into boolean satisfiability, on a set of histories generated by running random clients on realistic cache coherence protocols implemented in the Gem5 simulator [17]. Similarly to SC, we have used the following cache coherence protocols: MI, MEOSI Hammer, MESI Two Level, and MEOSI AMD Base. The parameters of these random clients are the number of cpus (threads) and the total number of read/write operations. We ensure that all the histories are differentiated i.e., writes on the same variable are unique.

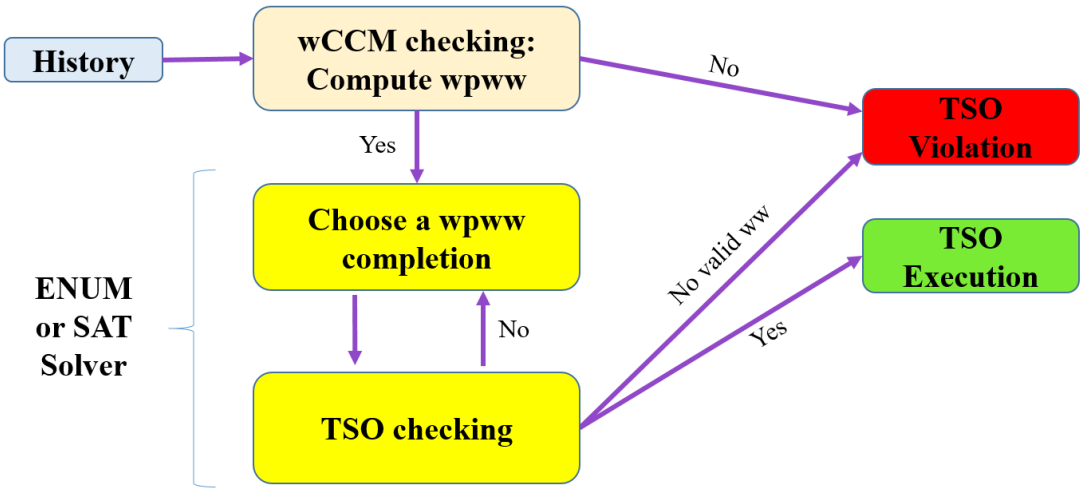


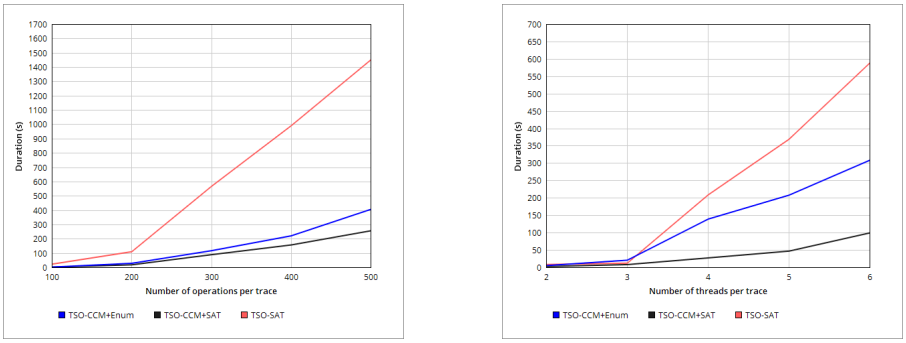
Figure 5.4 – The general schema of the TSO checking procedure using wCCM

We have compared two variations of our algorithms for checking TSO with a standard encoding of TSO into boolean satisfiability (named TSO-SAT). The two variations differ in the way in which the partial store order wpww imposed by wCCM is completed to a total store order ww as required by TSO: either using standard enumeration (named TSO-CCM+Enum) or using a SAT solver (named TSO-CCM+SAT).

Similarly to pww in the case of CCM, the computation of the partial store order wpww is done using an encoding of its definition into a DATALOG program. The inductive definition of wlhb_o supports an easy translation to DATALOG rules, and the same holds for the union of two relations, or their composition. The obtained Datalog programs were run using Clingo [39].

The Figure 5.4 shows the general schema of the testing procedure that we used in these experiments to check TSO using wCCM.

We have evaluated our TSO algorithms on the same set of histories used for SC in Figure 4.6. Since these histories satisfy SC, they satisfy TSO as well. Our algorithms scale much better than the SAT encoding. However, differently from SC, the enumeration of wpww extensions using a SAT solver outperforms the explicit enumeration. Since this difference was more negligible in the case of SC, it seems that the SAT variation is generally better.



(a) Checking TSO while varying the number of operations.

(b) Checking TSO while varying the number of cpus.

Figure 5.5 – Checking TSO for valid histories.

5.1.5 Discussion

We have presented an approach for verifying TSO by generalizing the first SC verification approach introduced in Section 4.1 (Chapter 4). The approach consists in using a weak consistency model called wCCM that is checkable in polynomial time to approximate TSO. This approach allows computing a set of orders between writes on the same variable that is included in any store order that witnesses for TSO conformance, if it exists. In addition, it allows detecting TSO violations early (i.e., violations that are already wCCM violations) and in only polynomial time. Our Experiment results show that using wCCM allows to improve the TSO checking performance w.r.t. an explicit enumeration using a reduction of the problem to SAT. Now, the question again is how far this approach can be pushed? or can we found a stronger consistency model that can be used to approximate TSO in a more efficient way?

The following section investigates this question.

5.2 Approach 2: wTSO-based TSO verification

In this section we introduce a new saturation-based consistency model called weak Total Store Ordering (wTSO) used to approximate TSO. First of all, we prove that wTSO is strictly stronger than wCCM and weaker than TSO. The experimental results show that wTSO scales much better than wCCM when both the number of threads and the number of operations increase. Section 5.2.1

introduces wTSO. Then, Section 5.1.2 describes our wTSO-based algorithm for checking TSO. The complexity of our approach is discussed in the Section 5.2.3. Finally, the experiment results are shown in the Section 5.2.4.

5.2.1 Weak Total Store Ordering

We propose a new consistency model called weak Total Store Ordering (wTSO). This new model computes a partial store order using a simpler *saturation rule* compared to the one used in wCCM.

Formally, for $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, let wst and whb^{π} be the smallest relations such that

$$\begin{aligned} \mathsf{wst} &= ((\mathsf{whb}^{\pi}{}_{\mathbb{WR}} \circ \mathsf{wr}^{-1}) \cup \mathsf{whb}^{\pi}{}_{\mathbb{WW}})^{+} \\ \mathsf{whb}^{\pi} &= (\pi \cup \mathsf{wr}_e \cup \mathsf{wst} \cup \mathsf{rw}[\mathsf{wst}])^{+} \\ \mathsf{rw}[\mathsf{wst}] &= \mathsf{wr}^{-1} \circ \mathsf{wst} \end{aligned}$$

Where, $\mathsf{whb}^{\pi}{}_{\mathbb{WW}}$, resp. $\mathsf{whb}^{\pi}{}_{\mathbb{WR}}$, is the projection of whb^{π}, resp. whb^{π}, on pairs of writes, resp. on pairs of writes and reads, on the same variable parameterized by π. The external write-read is $\mathsf{wr}_e = \mathsf{wr} \cap \{(o, o') \mid (o, o') \notin \mathsf{po} \text{ and } (o', o) \notin \mathsf{po}\}$. Then,

Definition 18 *A* history $\langle O, \mathsf{po}, \mathsf{wr}\rangle$ *satisfies weak Total Store Ordering (wTSO) if both* $\mathsf{whb}^{\mathsf{ppo}}$ *and* $\mathsf{whb}^{\mathsf{po\text{-}loc}}$ *are acyclic.*

To illustrate, consider the following examples,

Example 37 *The Figure 5.6a presents a history which is conform to wTSO. To show this, one can consider that the writes* $\mathsf{write}(z, 3)$ *and* $\mathsf{write}(z, 4)$ *are not related by the partial store order wst. Roughly, the wTSO saturation rules does not impose any order for this pair of writes* ($\mathsf{write}(z, 3)$ *and* $\mathsf{write}(z, 4)$).

Example 38 *The Figure 5.6b presents a history which does not satisfy wTSO. Since* $\mathsf{read}(y, 0)$ *in* t_2 *returns the initial value 0, it should precede all writes in the variable y, so* $\mathsf{read}(y, 0)$ *should precede* $\mathsf{write}(y, 1)$ *in read-write relation. Similarly, since* $\mathsf{read}(x, 0)$ *in* t_3 *returns the initial value 0, it should precede* $\mathsf{write}(x, 1)$ *in read-write relation* $\mathsf{rw}[\mathsf{wst}]$*. Then, we get a cycle in* $\mathsf{rw}[\mathsf{wst}] \cup \mathsf{wr}_e$.

t_0:	t_1:	t_2:	t_3:
write$(x,1)$	write$(x,2)$	read$(t,1)$	read$(t,1)$
write$(y,1)$	write$(y,2)$	write$(x,4)$	write$(y,4)$
write$(z,3)$	write$(z,4)$	write$(t,3)$	write$(t,4)$
write$(t,1)$	write$(t,2)$		
t_4:	t_5:	t_6:	t_7:
read$(t,2)$	read$(t,2)$	read$(t,3)$	read$(t,4)$
write$(x,3)$	write$(y,3)$	read$(y,2)$	read$(x,2)$
write$(t,5)$	write$(t,6)$		
	t_8:	t_9:	
	read$(t,5)$	read$(t,6)$	
	read$(y,1)$	read$(x,1)$	

(a) wTSO but not TSO

t_0:	t_1:	t_2:	t_3:
write$(x,1)$	write$(y,1)$	read$(x,1)$	read$(y,1)$
		read$(y,0)$	read$(x,0)$

(b) wCCM but not wTSO nor TSO.

Figure 5.6 – Comparison of wTSO and TSO consistency models.

We prove that wTSO is stronger than wCCM.

Lemma 16 *If a* history *satisfies wTSO, then it satisfies wCCM.*

Proof 13 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ *be a* history *which satisfies wTSO i.e.,* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{wst} \cup \mathsf{rw}[\mathsf{wst}]$ *and* $\mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{wst} \cup \mathsf{rw}[\mathsf{wst}]$ *are both acyclic. We show that for* $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, $(\pi \cup \mathsf{wr}_e \cup \mathsf{wpww} \cup \mathsf{rw}[\mathsf{wpww}])^+ \subseteq \mathsf{whb}^\pi$ *(the history satisfies wCCM as well). Let* $\mathsf{co}^\pi = (\pi \cup \mathsf{wr}_e)^+$.

Let's prove that $\mathsf{lhb}_o^\pi \subseteq \mathsf{whb}^\pi$ *for every operation* o *in* h. *To do, we prove that* whb^π *fulfill the two requirements of* lhb_o^π*:*

- *If* $(o_1, o_2) \in \mathsf{co}^\pi$, $(o_1, o) \in \mathsf{co}^\pi$, *and* $(o_2, o) \in \mathsf{co}^\pi$ *then* $(o_1, o_2) \in \mathsf{whb}^\pi$ *(the reason is that* $\mathsf{co}^\pi \subseteq \mathsf{whb}^\pi$*), and*
- *If* $(\mathit{write}(x,v), \mathit{read}(x,v')) \in \mathsf{whb}^\pi$ *and* $(\mathit{write}(x,v'), \mathit{read}(x,v')) \in \mathsf{wr}$ *then* $(\mathit{write}(x,v), \mathit{write}(x,v')) \in (\mathsf{whb}^\pi \circ \mathsf{wr}^{-1})$ *and then* $(\mathit{write}(x,v), \mathit{write}(x,v')) \in \mathsf{wst}$ *and* $(\mathit{write}(x,v), \mathit{write}(x,v')) \in \mathsf{whb}^\pi$.

Therefore, we get $\mathsf{lhb}_o^\pi \subseteq \mathsf{whb}^\pi$ *and then* $\mathsf{wlhb} \subseteq \mathsf{whb}^\pi$.

Let us now show that $\mathsf{wpww} = (\mathsf{wlhb}_{\mathbb{WW}} \cup \mathsf{cf}_e[\mathsf{lhb}^{\mathsf{po\text{-}loc}}] \cup \mathsf{cf}_e[\mathsf{lhb}^{\mathsf{ppo}}])^+ \subseteq \mathsf{wst}$.

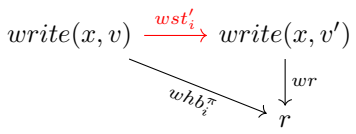


Figure 5.7 – Partial store order wst'_i used to define wTSO consistency model

It can be see that $\mathsf{wlhb}_{\mathbb{WW}} \subseteq \mathsf{whb}^{\pi}_{\mathbb{WW}}$ *(since* $\mathsf{wlhb} \subseteq \mathsf{whb}^{\pi}$ *for* $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$*). By definition,* $\mathsf{cf}_e[\mathsf{lhb}^{\pi}] = \mathsf{lhb}^{\pi}{}_{\mathbb{WR}} \circ \mathsf{wr}_e^{-1}$ *and then* $\mathsf{cf}_e[\mathsf{lhb}^{\pi}] \subseteq (\mathsf{whb}^{\pi}{}_{\mathbb{WR}} \circ \mathsf{wr}^{-1})$*. This implies that* $\mathsf{wpww} = (\mathsf{wlhb}_{\mathbb{WW}} \cup \mathsf{cf}_e[\mathsf{lhb}^{\mathsf{po\text{-}loc}}] \cup \mathsf{cf}_e[\mathsf{lhb}^{\mathsf{ppo}}])^+ \subseteq \mathsf{wst} = ((\mathsf{whb}^{\pi}{}_{\mathbb{WR}} \circ \mathsf{wr}^{-1}) \cup \mathsf{whb}^{\pi}{}_{\mathbb{WW}})^+$.

Finally, we can deduce that for $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, $(\pi \cup \mathsf{wr}_e \cup \mathsf{wpww} \cup \mathsf{rw}[\mathsf{wpww}])^+ \subseteq \mathsf{whb}^{\pi} = (\pi \cup \mathsf{wr}_e \cup \mathsf{wst} \cup \mathsf{rw}[\mathsf{wst}])^+$. □

The reverse of this lemma does not hold. Take the following example for instance,

Example 39 *Figure 5.6b presents a* history *that satisfies wCCM but not wTSO (the reason was explained before in the examples 32 and 38).*

Then,

Lemma 17 *wTSO is strictly stronger than wCCM.*

We show now that wTSO is weaker than TSO. Let's define sub relations of wst and whb^{π} that are gotten using iterative least fix-point computation.

Let $\mathsf{wst} = \bigcup_i wst_i$ and for $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, $\mathsf{whb}^{\pi} = \bigcup_i whb_i^{\pi}$ where $wst_i = (\mathsf{whb}_{\mathsf{i}}^{\pi}{}_{\mathbb{WW}} \cup wst'_i)^+$ and wst'_i (Figure 5.7) is defined as follows:

$$(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in wst'_i \text{ iff } (\mathsf{write}(x, v), \mathsf{read}(x, v')) \in whb_i^{\pi} \text{ and } (\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}$$

where, for $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$ and for every $i \geq 0$, whb_i^{π} is defined by:

$$\begin{aligned} whb_0^{\pi} &= (\pi \cup \mathsf{wr}_e)^+ \\ whb_{i+1}^{\pi} &= (whb_i^{\pi} \cup wst_i^{\pi} \cup \mathsf{rw}[wst_i^{\pi}])^+ \end{aligned}$$

We now prove that the partial store order wst_i is a part of any store order ww that witnesses for TSO conformance.

Lemma 18 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ *be a* history *and* ww *be a total store order such that* $\mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ *and* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ *are both acyclic. Then,* $wst_i \subseteq \mathsf{ww}$ *and* $whb_i^\pi \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$.

Proof 14 *Let* $h = \langle O, \mathsf{po}, \mathsf{wr}\rangle$ *be a* history *that is conform to TSO i.e., there exists a total order* ww *such that (*$\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$*) and (*$\mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$*) are acyclic. We prove that, for* $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, $whb_i^\pi \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *and* $wst_i^\pi \subseteq \mathsf{ww}$ *for all* ww *such that* $(\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})$ *is acyclic. We show this using the induction on the index* i *of* whb_i^π *and* wst_i.
Base-Case. *The* $whb_0^\pi = (\pi \cup \mathsf{wr}_e)^+$ *is included in* $(\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$. *Considering* $whb_0^\pi \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$, *if* $(\mathsf{write}(x, v), \mathsf{read}(x, v')) \in whb_0^\pi$ *and there exists a* $\mathsf{read}(x, v')$ *such that* $(\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}$, *then* $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{ww}$. *Otherwise, assuming by contradiction that* $(\mathsf{write}(x, v'), \mathsf{write}(x, v)) \in \mathsf{ww}$, *then* $(\mathsf{read}(x, v'), \mathsf{write}(x, v)) \in \mathsf{rw}$. *On the other hand, considering* $\mathsf{write}(x, v), \mathsf{read}(x, v')) \in whb_0^\pi \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$, *we get a cycle in* $(\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *which is a contradiction. Therefore,* $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{ww}$.

Thus, wst'_0 *is included in* ww, *then* $wst_0 = (\mathsf{whb}_{0\,\mathbb{WW}}^\pi \cup wst'_0)^+$ *is also included in* ww *(the reason is that* $\mathsf{whb}_{0\,\mathbb{WW}}^\pi \subseteq \mathsf{ww}$*). Otherwise, it leads to a contradiction with the fact that* $whb_0^\pi \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *and* $(\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *is acyclic for* $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$.

Induction Step. *Suppose that for all* ww, $whb_i^\pi \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *and* $wst_i \subseteq \mathsf{ww}$. *We prove that this holds for* $i + 1$ *as well. By induction hypothesis,* $wst_i \subseteq \mathsf{ww}$, *so using the definition of* $\mathsf{rw}[wst_i]$ *we have* $\mathsf{rw}[wst_i] \subseteq \mathsf{rw}$. *Then,* $whb_{i+1}^\pi = (whb_i^\pi \cup wst_i \cup \mathsf{rw}[wst_i])^+ \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$. *Now, let's prove that* $wst'_{i+1} \subseteq \mathsf{ww}$. *If* $(\mathsf{write}(x, v), \mathsf{read}(x, v')) \in whb_i^\pi$ *and* $(\mathsf{write}(x, v'), \mathsf{read}(x, v')) \in \mathsf{wr}$, *then* $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{ww}$. *Otherwise, using the same argument above (in the base case),* $(\mathsf{read}(x, v'), \mathsf{write}(x, v)) \in \mathsf{rw}$ *which is a contradiction with the fact that* $(\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *is acyclic. Therefore, if* $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in wst'_{i+1}$ *then* $(\mathsf{write}(x, v), \mathsf{write}(x, v')) \in \mathsf{ww}$ *and then* $wst'_{i+1} \subseteq \mathsf{ww}$. *Moreover,* $\mathsf{whb}_{\mathsf{i+1}\,\mathbb{WW}}^\pi \subseteq \mathsf{ww}$ *since* $whb_{i+1}^\pi \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *(Otherwise it leads to a contradiction with the fact that* $(\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *is acyclic). Since* $wst_{i+1} = (\mathsf{whb}_{\mathsf{i+1}\,\mathbb{WW}}^\pi \cup wst'_{i+1})^+$, $wst'_{i+1} \subseteq \mathsf{ww}$ *and* $\mathsf{whb}_{\mathsf{i+1}\,\mathbb{WW}}^\pi \subseteq \mathsf{ww}$, *we get* $wst_{i+1} \subseteq \mathsf{ww}$ *(since* ww *is a total store order).* □

Then, as an immediate corollary of Lemma 18, we get:

Lemma 19 *If a* history *satisfies TSO, then it satisfies wTSO.*

Proof 15 *The proof is by contradiction. Assume that a history* $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$ *satisfies TSO and it does not satisfy wTSO.*

The history h *is conform to TSO means that there exists a total store order* ww *such that* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ *and* $\mathsf{po\text{-}loc} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$ *are acyclic. Considering that* h *does not satisfy wTSO i.e.,* whb^{ppo} *or* $whb^{\mathsf{po\text{-}loc}}$ *or both of them are cyclic. Since, for* $\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$, $\mathsf{whb}^{\pi} = \bigcup_i whb_i^{\pi}$ *and* $whb_i^{\pi} \subseteq (\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *(from Lemma 18), we can deduce that* $(\pi \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw})^+$ *is also cyclic and thus we get a contradiction.* □

The reverse of the above lemma doesn't hold. For instance,

Example 40 *Figure 5.6 shows a* history *which satisfies wTSO (The reason was shown in example 37) but not TSO. Since there is no valid store order for the writes* $\mathsf{write}(z, 3)$ *and* $\mathsf{write}(z, 4)$*, this* history *does not satisfy TSO. In fact, consider the two possible cases:*

- *If* $\mathsf{write}(z, 3)$ *precedes* $\mathsf{write}(z, 4)$ *in* ww*, then* $\mathsf{write}(y, 1)$ *precedes* $\mathsf{write}(y, 3)$ *in* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww}$*. Since,* $\mathsf{read}(y, 1)$ *reads its value from* $\mathsf{write}(y, 1)$*, we get a* rw *between* $\mathsf{read}(y, 1)$ *and* $\mathsf{write}(y, 3)$*. Similarly,* $\mathsf{write}(x, 1)$ *precedes* $\mathsf{write}(x, 3)$ *in* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww}$ *and* $\mathsf{read}(x, 1)$ *reads its value from* $\mathsf{write}(x, 1)$*, then, we get a* rw *between* $\mathsf{read}(x, 1)$ *and* $\mathsf{write}(x, 3)$*. Thus, we get a cycle in* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$*.*
- *If* $\mathsf{write}(z, 4)$ *happens before* $\mathsf{write}(z, 3)$ *in* ww*, then* $\mathsf{write}(y, 2)$ *precedes* $\mathsf{write}(y, 4)$ *in* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww}$*. Since,* $\mathsf{read}(y, 2)$ *reads its value from* $\mathsf{write}(y, 2)$*, we get a* rw *between* $\mathsf{read}(y, 2)$ *and* $\mathsf{write}(y, 4)$*. Similarly, we get a* rw *between* $\mathsf{read}(x, 2)$ *and* $\mathsf{write}(x, 4)$*. Thus, we get a cycle in* $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$*.*

Both of these cases lead to a cycle in $\mathsf{ppo} \cup \mathsf{wr}_e \cup \mathsf{ww} \cup \mathsf{rw}$*, so the history is not allowed under TSO model.*

Then,

Lemma 20 *TSO is strictly stronger than wTSO.*

The Figure 5.8 presents the whole image of the relationships between the consistency models studied in this thesis. In this chapter, we have introduced

wCCM which is strictly weaker than CCM, incomparable with CM and strictly weaker than TSO. We have also introduced wTSO that is strictly stronger than wCCM and strictly weaker than TSO.

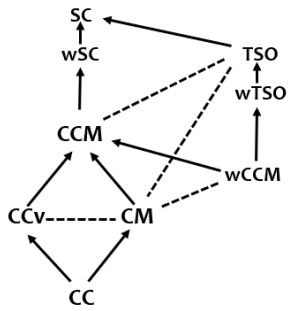


Figure 5.8 – Relationships between all consistency models considered in this thesis.

5.2.2 An Algorithm for checking TSO conformance using wTSO

In this section, we define an algorithm for checking TSO conformance which completes the partial store order wst, computed by the wTSO saturation rules, to a total store order. Following the SC verification approaches and the TSO verification approach that we have presented in Section 5.1, we start by checking that the given history satisfies wTSO. If not, then we conclude that it is not TSO as well (by Lemma 19). If yes, we exploit wst in order to enhance the TSO verification of the history. Then, we use the SAT solver to find a total store order extending wst. It starts enumerating the orders between the writes that are not related by the wst order until it founds one that allows establishing TSO satisfaction, so in this case it returns true. Otherwise, it returns false. Hence,

Theorem 11 *Algorithm 7 returns true iff the input history h satisfies TSO.*

5.2.3 Complexity

Since at each step of the computation of whb^{π} ($\pi \in \{\mathsf{ppo}, \mathsf{po\text{-}loc}\}$) and wst, we enumerate over three operations ($\mathcal{O}(n^3)$) and at least one pair of operations is

Algorithm 7: Checking TSO conformance: wTSO+ENUM algorithm.

Input: A history $h = \langle O, \mathsf{po}, \mathsf{wr} \rangle$
Output: *true* iff h satisfies TSO

```
1 if ppo ∪ wr_e ∪ wst ∪ rw[wst] or po-loc ∪ wr_e ∪ wst ∪ rw[wst] is cyclic then
2     return false;
3 foreach ww ⊃ wst do
4     if ppo ∪ wr_e ∪ ww ∪ rw[ww] and po-loc ∪ wr_e ∪ ww ∪ rw[ww] are acyclic then
5         return true;
6 return false;
```

added to one of these two relations and the number of such pairs is polynomially bounded (in the size of the computation), at most n^2 edges where n is the computation size. Then, the acyclicity of whb^{π} can be decided in polynomial time. Hence,

Theorem 12 *Checking whether a* history h *satisfies wTSO is polynomial time* ($\mathcal{O}(n^5)$) *in the size of the* history.

5.2.4 Experimental Evaluation

The Figure 5.9 presents the approach that we propose to check TSO conformance based on wTSO.

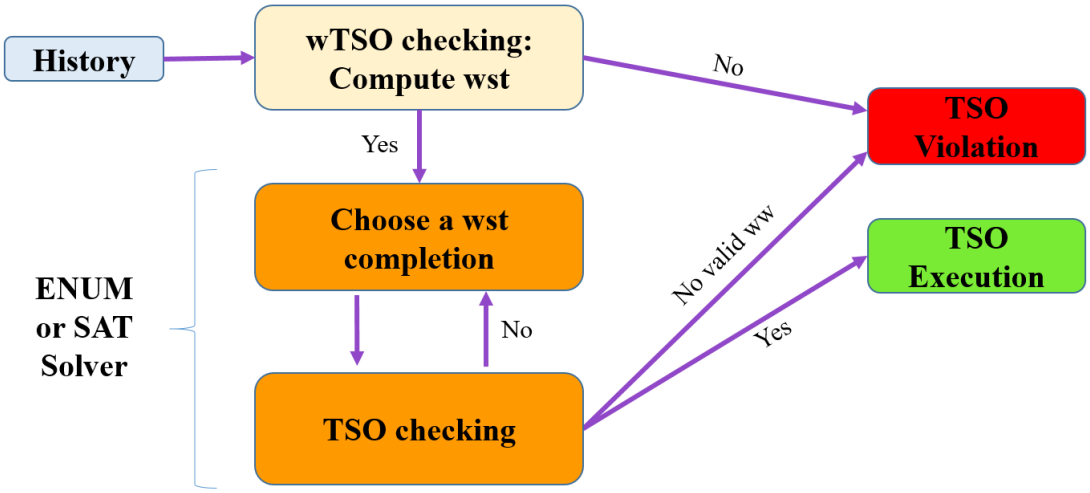
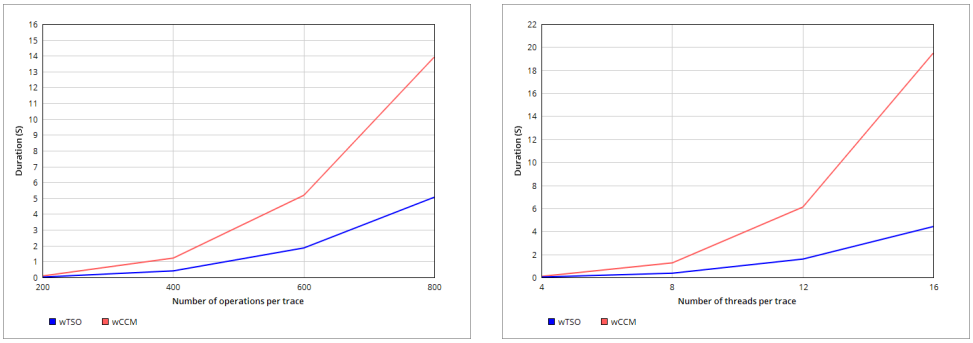


Figure 5.9 – The general schema of the TSO checking procedure using wTSO

Since wTSO is strictly stronger than wCCM, we believe that the implementation of this procedure (Figure 5.9) will outperform the one based on wCCM.



(a) Checking wTSO and wCCM while varying the number of operations.

(b) Checking wTSO and wCCM while varying the number of cpus.

Figure 5.10 – Checking wTSO and wCCM for valid histories.

The reason is that in addition of capturing more violations using wTSO (because wTSO is strictly stronger than wCCM), it allows computing a large subset of the total order witnessing for TSO conformance, if any. The adaptation of DBCOP to the case of TSO is non-trivial. This is left for future work.

We have implemented the wTSO and wCCM checking algorithms and compare them using a set of histories generated using Gem5 [17]. The Figure 5.10 presents the results of this comparison. The figure 5.10a presents the effect of increasing the number of operations from 200 to 800 on runtime for a fixed number of processes (6 processes per trace). The figure 5.10b shows the effect of increasing the number of processes from 4 to 16. We have tested 150 histories for each case and computed the average runtime.

As expected, the results shown in Figure 5.10 confirm that wTSO has better performance compared to wCCM, by factors of 5 times in the case of 16 threads (Figure 5.10b). Notice that all the histories that we have tested were valid w.r.t. wTSO and wCCM.

5.3 Conclusion

We have introduced two approaches for tackling the problem of checking TSO conformance. The idea is to avoid falling in a systematic way in the worst case i.e., an explicit enumeration of the exponential number of possible total orders between writes. These approaches consists in defining weaker consistency models that are as strong as possible but still polynomial time checkable. These models

allow capturing a large subset of the partial order on writes which can be computed using a least fix point calculation (i.e., in polynomial time), and that is a subset of any total order witnessing TSO conformance.

CHAPTER 6

CONCLUSIONS

In this final chapter, we summarize the results of this thesis and discuss some possible extensions of the works presented in the chapters above.

Summary

In this thesis, we focused in the verification problem of concurrent/distributed systems. The main contribution is the proposition of an efficient approach for verifying consistency which is generically applicable to a wide spectrum of consistency models for which the problem of verification is known to be NP-hard. Indeed, we proposed dynamic verification approaches with respect to some well known consistency models.

The first part of this thesis was dedicated to the verification of weak consistency models e.g., causal consistency in its different variants: weak causal consistency, causal convergence, and causal memory. This work is based on a characterization of the set of all histories that are causal consistency violations called bad-patterns. These bad-patterns define a small set of operations occurring in some particular order. We proposed a polynomial reduction of these bad-patterns to a problem of solving Datalog queries. The proposed approach allows to improve the complexity of checking causal consistency problem (from $\mathcal{O}(n^5)$ to $\mathcal{O}(n^3)$). The experiments on real distributed databases showed the efficiency and scalability of the proposed approach.

The second part of the thesis focused on the verification of strong consistency models e.g., Sequential consistency (SC) and Total Store Ordering models (TSO is a relaxation of SC).

First of all, we built on the causal consistency verification approach to define a gradual approach for checking SC. In fact, this approach is based on a strong variant of causal consistency (stronger than weak causal consistency, causal convergence, and causal memory) called convergent causal memory (CCM) which allows efficiently approximating SC model. Afterwards, we extended this approach to define a more simpler and natural approximation for SC called weak sequential consistency (wSC). The experiment results, obtained using realistic cache coherence protocols, showed that both approaches perform good results compared to an encoding of SC into boolean satisfiability. Furthermore, the wSC-based approach outperforms the CCM-based approach. This is due to the fact that wSC is stronger than CCM and to the fact that CCM uses more complicated saturation rules compared to the simpler ones used in wSC. A combination of wSC with an existent approach called DBCOP led to the best results.

Second, we addressed the problem of verifying the TSO model which is weaker than SC. In fact, these SC verification approaches were generalized to cover the TSO case. Since SC and TSO use different relations, the generalization of these approaches was not trivial. Our focus in this stage was defining suitable approximations for TSO. Indeed, we defined two TSO verification approaches, one based on a criterion called wCCM and another based on wTSO criterion. Similarly to the SC case, the experiments performed using real cache coherence protocols proved that the two approaches are more efficient and more scalable compared to the standard SAT encoding of TSO. In addition, the wTSO has better performances compared to wCCM.

Future Work

In terms of future work, several extensions of the work we have introduced in this thesis are possible. Next, we summarized some possible directions.

1. In chapter 3, we proposed a reduction of the causal consistency verification problem to a problem of solving Datalog queries. It will be interesting to see how a similar approach can be used to solve the verification problem

for other polynomially checkable consistency models such as consistency models for transactions i.e., Read Committed, Read Atomic, and Causal consistency for transactions.

2. Since the approach we proposed to check strong consistency models was applicable for two consistency models SC and TSO (Chapter 4 and Chapter 5), an interesting problem for future work is the application of this approach to other correctness criteria that are hard to check (the problem of verification is NP-hard). The consistency models for transactions are good candidates to explore i.e., Prefix consistency, Snapshot isolation, and Serializability. Actually, the latter is the analogous of Sequential consistency in transactional programs context. Therefore, it can be a starting point for an eventual adaptation.
3. Another direction could be the adaptation of DBCOP to the case of TSO. The idea is to propose a similar approach to BTTC (defined in [7] and [18]), which is efficient in the case of a fixed number of threads. In addition, the combination of a such approach with wTSO may result a more efficient approach for checking TSO.
4. In Section 4.2.2 of the chapter 4, we introduced the notion of SC kernel (SC-Ker) and we proved some interesting results about it. One of these results is that neither wSC nor an extended saturation procedure can calculate the SC-Ker. So, a question for future work would be whether there is a way for computing the SC-kernel of a given SC execution in polynomial time. Furthermore, it will be interesting to explore the TSO kernel as well.

BIBLIOGRAPHY

[1] http://jepsen.io. [Retrieved 01/12/2020.].

[2] https://potassco.org/clingo/run/. [Retrieved 26/12/2020.].

[3] https://www.cockroachlabs.com. [Retrieved 15/11/2018].

[4] http://galeracluster.com. [Retrieved 15/11/2018].

[5] https://github.com/codership/galera/issues/336. [Retrieved 15/11/2018].

[6] https://www.gem5.org/documentation/general_docs/debugging_and_testing/directed_testers/ruby_random_tester/. [Retrieved 10/12/2020.].

[7] 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. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

[8] Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. Parameterized verification through view abstraction. *STTT*, 18(5):495–516, 2016.

[9] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[10] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distrib. Comput.*, 9(1):37–49, March 1995. doi:10.1007/BF01784241.

[11] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[12] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[13] Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.

[14] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2463676.2465279, doi:10.1145/2463676.2465279.

[15] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[16] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[17] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[18] Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

[19] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 626–638. ACM, 2017. URL: http://dl.acm.org/citation.cfm?id=3009888.

[20] Sebastian Burckhardt. *Principles of Eventual Consistency*. now publishers, 2014.

[21] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 330–340. ACM, 2010.

[22] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989. doi:10.1109/69.43410.

[23] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *CHDL*, volume A-32 of *IFIP Transactions*, pages 15–30. North-Holland, 1993.

[24] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, page 176–194, Berlin, Heidelberg, 2001. Springer-Verlag.

[25] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[26] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. doi:10.1145/512950.512973.

[27] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[28] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In *CAV*, volume 1855 of *LNCS*, pages 53–68. Springer, 2000.

[29] Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[30] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2523616.2523628, doi:10.1145/2523616.2523628.

[31] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. *Proceedings of the 5th ACM Symposium on Cloud Computing, SOCC 2014*, 11 2014. doi:10.1145/2670979.2670983.

[32] Ásgeir Th. Eiríksson and Kenneth L. McMillan. Using formal verification/-analysis methods on the critical path in system design: A case study. In Pierre Wolper, editor, *CAV*, volume 939 of *LNCS*, pages 367–380. Springer, 1995.

[33] Marco Elver and Vijay Nagarajan. Mcversi: A test generation framework for fast memory consistency verification in simulation. In *HPCA*, pages 618–630. IEEE Computer Society, 2016.

[34] Michael Emmi and Constantin Enea. Monitoring weak consistency. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 487–506. Springer, 2018.

[35] Michael Emmi and Constantin Enea. Sound, complete, and tractable linearizability monitoring for concurrent collections. *PACMPL*, 2(POPL):25:1–25:27, 2018.

[36] Michael Emmi, Constantin Enea, and Jad Hamza. Monitoring refinement via symbolic reasoning. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 260–269. ACM, 2015.

[37] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999.

[38] Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embedded Comput. Syst.*, 14(4):63:1–63:25, 2015.

[39] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014. URL: http://arxiv.org/abs/1405.3694, arXiv:1405.3694.

[40] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[41] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997.

[42] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. URL: http://doi.acm.org/10.1145/564585.564601, doi:10.1145/564585.564601.

[43] Alexey Gotsman and Sebastian Burckhardt. Consistency models with global operation sequencing and their composition. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.23.

[44] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[45] Ernesto Jiménez, Antonio Fernández Anta, and Vicent Cholvi. A parametrized algorithm that implements sequential, causal, and cache memory consistencies. *Journal of Systems and Software*, 81:120–131, 01 2008. doi:10.1016/j.jss.2007.03.012.

[46] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. URL: http://doi.acm.org/10.1145/359545.359563, doi:10.1145/359545.359563.

[47] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[48] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2043556.2043593, doi:10.1145/2043556.2043593.

[49] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4), 2017. doi:10.1002/cpe.3928.

[50] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, convergence. Technical report, 2011.

[51] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, USA, 1992. UMI Order No. GAX92-24209.

[52] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[53] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276530.

[54] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. Causal consistency: beyond memory. In *PPoPP*, pages 26:1–26:12. ACM, 2016.

[55] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. *SIGOPS Oper. Syst. Rev.*, 31(5):288–301, October 1997. URL: http://doi.acm.org/10.1145/269005.266711, doi:10.1145/269005.266711.

[56] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Trans. Parallel Distrib. Syst.*, 6(8):773–787, 1995.

[57] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, SRDSW '14, pages 30–33, Washington, DC, USA, 2014. IEEE Computer Society. doi:10.1109/SRDSW.2014.33.

[58] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Trans. Parallel Distrib. Syst.*, 14(8):730–741, 2003.

[59] Amitabha Roy, Stephan Zeisset, Charles J. Fleckenstein, and John C. Huang. Fast and generalized polynomial time memory consistency verification. In *CAV*, volume 4144 of *LNCS*, pages 503–516. Springer, 2006.

[60] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. doi:10.1145/1785414.1785443.

[61] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 137–146, New York, NY, USA, 1982. ACM. URL: http://doi.acm.org/10.1145/800070.802186, doi:10.1145/800070.802186.

[62] David S Warren. Programming in tabled prolog. In *Symposium Program*, page 62, 1999.

[63] Jeannette M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2):164–182, 1993. doi:10.1006/jpdc.1993.1015.

[64] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL*, pages 184–193. ACM Press, 1986.

[65] Rachid Zennou, Mohamed Faouzi Atig, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Boosting sequential consistency checking using saturation. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 360–376, Cham, 2020. Springer International Publishing. URL: https://doi.org/10.1007/978-3-030-59152-6_20.

[66] Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Checking causal consistency of distributed databases. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems*, pages 35–51, Cham, 2019. Springer International Publishing. URL: https://doi.org/10.1007/978-3-030-31277-0_3.

[67] Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Checking causal consistency of distributed databases. 2020. URL: https://arxiv.org/abs/2011.09753.

[68] Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Checking causal consistency of distributed databases. *Computing*, February 2021. doi:10.1007/s00607-021-00911-3.

[69] Rachid Zennou, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Gradual consistency checking. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 267–285, Cham, 2019. Springer International Publishing. URL: https://doi.org/10.1007/978-3-030-25543-5_16.

APPENDIX A

SYNTHÈSE DE LA THÈSE EN FRANÇAIS

L'évolution de notre société moderne, basée sur le développement spectaculaire des technologies de l'information et de la communication, est étroitement liée avec notre besoin croissant de services automatisés qui sont devenus cruciaux dans tous les secteurs de notre vie (communication, commerce, finance, transport, santé, loisirs, énergie, etc). Avec l'émergence de l'Internet of Things et du Cloud Computing, il y aura de plus en plus d'objets connectés de toutes sortes, communiquant et interagissant à travers des réseaux à grande échelle, ayant accès à des ressources en mémoire et en puissance de calcul virtuellement illimitées. Le déploiement de ces systèmes fortement distribués et la maitrise de leur complexité posent d'énormes défis scientifiques et technologiques. La quête de la performance pousse les concepteurs et les développeurs des systèmes informatiques à avoir recours à différentes sortes d'optimisations, et en particulier à de plus en plus de parallélisation et de distribution, avec un usage parcimonieux de la synchronisation. L'idée générale est de chercher à augmenter le débit du système, de rendre les données disponibles et rapidement accessibles aux clients, et d'éviter les attentes dues aux actions bloquantes. Cela se produit en fait à tous les niveaux des systèmes informatiques, du niveau le plus bas qui est celui des architectures matérielles multi-cœurs au niveau le plus élevé qui est celui des applications réparties qui s'exécutent sur des infrastructures en réseau, y compris les bases de données distribuées géo-répliquées. Ces optimisations ainsi que le caractère distribué des calculs, tendent à réordonner les actions effectuées par

chacune des composantes d'un système. Cela peut être dû par exemple au fait que les actions les plus coûteuses sont repoussées, ou exécutées en parallèle, afin de permettre aux actions les plus rapides (ou urgentes) d'être exécutées en premier. Cela peut aussi être dû à la latence des communications et au fait que les messages peuvent suivre des chemins différents à travers des réseaux de grande taille. Un système peut alors avoir des comportements vis -à-vis des programmes clients qui ne sont pas possibles lorsque toutes les actions du système sont exécutées instantanément et de manière atomique, et sont visibles immédiatement à tous les processus dans le système, ce qui correspond au modèle dit de "consistance forte" (strong consistency). En fait, la consistance forte est en général difficile à assurer de manière acceptable du point de vue de la performance, et la majorité des systèmes utilisés en pratique (aussi bien les micro-processeurs modernes, que les plateformes pour le Cloud Computing) implémentent des modèles de consistance faibles. L'affaiblissement des garanties de consistance qu'un système assure aux programmes clients peut affecter la correction de ces derniers. Par exemple, si un système implémentant une base de données distribuée n'est pas fortement consistent, cela implique que les informations sur les différents repliques peuvent être différentes de certains instants puisque les mises à jours ne sont pas immédiatement visibles partout dans le système. Ceci peut impacter la correction des applications utilisant ce système aussi bien du point de vue de la sûreté que de la sécurité. Par exemple, deux opérations sur un compte pourraient s'effectuer en parallèle et retirer deux fois la même somme disponible avant leurs executions si les mises à jour ne sont pas visibles immédiatement. Aussi, si les politiques d'accès aux informations dans une base de données distribuée sont mises à jour de manière faiblement consistante, ce qui veut dire qu'elles peuvent être différentes à un moment donné d'un site à l'autre, cela peut entrainer des fuites d'informations qui sont censées être protégées.

Un des problèmes importants est alors d'assurer la correction de programmes (clients) qui vont s'exécuter sur des infrastructures qui implémentent un modèle de consistance faible. En effet, les systèmes concurrents et distribués sont notoirement difficiles à concevoir et à programmer de manière correcte. Cela est dû au grand nombre et à la complexité des interactions entre leurs composantes. Cette difficulté est d'autant plus grande lorsque ces systèmes doivent être exécutés selon un modèle de consistance faible qui permet encore plus de comportements

complexes, non intuitifs, et extrêmement difficiles à appréhender. Il est donc important de disposer de méthodes et d'outils pour la vérification automatique de programmes concurrents sur des modèles de consistance faibles, tenant compte aussi bien des propriétés de sureté que des propriétés de sécurité. Un deuxième problème important est celui de vérifier qu'un système qui est supposé assurer un service selon un modèle de consistance donné, est correctement implémenté. En effet, il est important de vérifier que les garanties en consistance (pour les clients) sont bien assurées par l'implémentation, ceci est un problème crucial, notamment en ce qui concerne les librairies d'objets et de structures de données distribuées géo-répliquées, qui sont les briques de base pour la construction des infrastructures modernes pour le Cloud Computing. Le but de cette thèse est d'étudier les deux problèmes de vérification mentionnés ci-dessus et d'apporter des solutions générales et efficaces pour les résoudre. Les solutions proposées sont génériquement applicables à un large spectre de modèles de consistance, en particulier ceux adoptés pour le raisonnement sur les systèmes distribués avec réplication. Nous résumons brièvement les contributions de notre thèse:

- Premièrement, nous avons considéré le problème de vérifier qu'une execution est conforme à un modèle de consistance faible. Nous avons proposé une approche pour vérifier les modèles de consistance causale en utilisant une réduction polynomiale de ce problème au problème de la résolution des requêtes Datalog. Par la suite, nous avons implémenté notre approche dans un outil efficace pour tester les systèmes distribués.
- Ensuite, nous avons abordé le problème de la vérification des modèles de la consistance forte. Nous avons considéré le modèle fondamental connu sous le nom de la consistance séquentielle (SC) et nous avons proposé une approche graduelle pour vérifier la conformité d'un execution donnée à ce modèle. Cette approche est basée sur un renforcement de tous les modèles de la consistance causale connus qui est encore vérifiable en temps polynomial. Ensuite, nous avons amélioré cette approche en proposant une autre approximation de SC, qui est plus naturelle et plus efficace et qui est également vérifiable en temps polynomial.
- Enfin, nous avons considéré le problème de la vérification d'un autre modèle de la consistance forte appelé "Total Store ordering" (TSO), qui est un affaiblissement de SC. En effet, nous avons généralisé les approches SC pour

couvrir le modèle TSO en proposant des modèles adaptés pour approximer TSO. Les généralisations suggérées des approches mentionnées ci-dessus ne sont pas triviales. En particulier, vu que ces deux modèles (SC et TSO) considèrent différents types de relations, ce dernier affaibli certaines relations considérées dans le premier.

Rachid ZENNOU

Email : rachid.zennou@gmail.com

LinkedIn: https://www.linkedin.com/in/rachid-zennou-323740a7/

Docteur en informatique

Formations et Diplômes

2017-2021 : Doctorat en informatique, université UM5 de Rabat, Maroc et université de Paris, France.

Sujet : **Méthodes algorithmiques pour la vérification de la consistance dans les systèmes distribués**.

Dirigée par : Pr. Mohammed Erradi (ENSIAS, Université Mohammed V Rabat) et

Pr. Ahmed Bouajjani (IRIF, Université de Paris)

2014-2016 : Master en Cryptographie et Sécurité de l'Information, Faculté des Sciences Rabat,
Mention : Bien.

2013-2014 : Licence en Sciences et Techniques, Faculté des Sciences et techniques Er-Rachidia,
Option : Génie Informatique, **Mention Bien**, **(Majorant de la promotion).**

Compétences

- **Systèmes distribués :** Spécifications, vérification et sécurité.
- **Cryptographie :** Crypto-system symétrique (AES, DES), asymétriques (RSA, El Gamal), signature électronique, Quantique BB84, Chaotique.
- **Sécurité informatique :** Sécurité (Réseau, Web et Système), Management de la sécurité (ISO 2700X, EBIOS), Audit sécurité (Pentest, Organisationnel, de configuration…).
- **Langages de programmation** : VB.NET, C, C++, Java, Python.
- **Machine Learning:** Weka, Scikit Learn, Matplotlib.
- **Certifications:** **ISO27001 Lead Implementer** & **ISO27005 Risk Manager**.
- **Divers:** OS (Windows, Linux), Suite Ms Office, Latex, Jira, Ms Project…

Prix et Excellence

- **Juin 2019: Best Student Paper Award**. Émetteur de la distinction: NETYS 2019 Program Committee, pour notre article: Checking Causal Consistency of Distributed Databases.
  Lien : http://netys.net/awards/
- **Août 2018: Grant** pour "Marktoberdorf Summer School" (Engineering Secure and Dependable Software Systems).
- **Mars 2018 : Bourse d'excellence** Ministère de l'Europe et des Affaires étrangères (attribuée pour les étudiants étrangers en fonction de l'excellence du candidat, telle qu'elle ressort de son parcours universitaire antérieur et pour le doctorat, ainsi que le caractère innovant de son sujet de recherche).

- **Juin 2017 : Bourse d'excellence CNRST** Maroc (attribuée sur la base de la qualité du cursus du candidat).
- **Décembre 2014 : Prix d'excellence** pour la majoration de la promotion en Licence à la Faculté des sciences et techniques à Er-Rachidia, Maroc.

Formations Supplémentaires :

- Participation au workshop **FoMLAS 2019**: Formal Methods for ML-Enabled Autonomous Systems Affiliated with CAV 2019, New-York, USA.
- Participation à l'école d'hiver « **VMCAI Winter School »** sur le thème **« Fundamental aspects of formal methods and applications »**, 9-12 Janvier, 2019, Lisbonne, Portugal.
- Participation à l'école d'été **Marktoberdorf Summer School** sur le thème **« Engineering Secure and Dependable Software Systems »**, 31 Juillet -11 Aout 2018, Munich, Allemagne.
- Participation au **Workshop sur la vérification des systèmes distribués**, **VDS'2018,** 6-8 Mai 2018, Essaouira, Maroc.
- Participation à l'école d'été **Marktoberdorf Summer School** sur le thème **« Logical Methods for Safety and Security of Software Systems »**, 2-11 Aout 2017, Munich, Allemagne.
- Participation à l'école d'été en **Théorie des jeux**, 01 - 05 juillet 2017 à l'ENSIAS, Rabat, Maroc.
- Participation à des formations sur les **communications scientifiques** et la **méthodologie de la recherche**, 2017, ENSIAS, Rabat.
- Participation à la conférence internationale **NETYS'2017, NETYS'2018, 2019** : International Conference on NETworked sYStems, Maroc.
- Participation à l'école de printemps **METIS'2017, METIS'2018** : International Spring School on Distributed Systems, Maroc.

Animations scientifiques :

- Membre de comité d'organisation de l'école de printemps « **The International Spring School on Distributed Systems (METIS)** » depuis 2017 jusqu'à 2019.
- Membre de comité d'organisation de la conférence « **The International Conference on Networked Systems (NETYS)** » depuis 2017 jusqu'à 2019.
- Membre de comité d'organisation de la 6 ème édition des « **Ateliers de communication et la journée interculturelle Maroc-Américaine »** en 2014 à la FST Er-rachidia, Maroc.

Publications scientifiques

- Zennou, R., Biswas, R., Bouajjani, A. et al. **Checking causal consistency of distributed databases**. Computing (2021). [Extended version of NETYS'2019 paper]. https://doi.org/10.1007/s00607-021-00911-3
- Zennou R., Atig M.F., Biswas R., Bouajjani A., Enea C., Erradi M. (2020) **Boosting Sequential Consistency Checking Using Saturation**. In: Hung D.V., Sokolsky O. (eds) Automated

Technology for Verification and Analysis. **ATVA 2020** (**classe A**). Lecture Notes in Computer Science, vol 12302. Springer, Cham. https://doi.org/10.1007/978-3-030-59152-6_20

- Zennou R., Bouajjani A., Enea C., Erradi M. (2019) **Gradual Consistency Checking**. In: Dillig I., Tasiran S. (eds) Computer Aided Verification. **CAV 2019 (classe A*).** Lecture Notes in Computer Science, vol 11562. Springer, Cham. https://doi.org/10.1007/978-3-030-25543-5_16
- Zennou R., Biswas R., Bouajjani A., Enea C., Erradi M. (2019) **Checking Causal Consistency of Distributed Databases**. In: Atig M., Schwarzmann A. (eds) Networked Systems. NETYS 2019. Lecture Notes in Computer Science, vol 11704. Springer, Cham. **[Best Student Paper Award]** https://doi.org/10.1007/978-3-030-31277-0_3

Langues

Tamazight : Maternelle **Arabe :** Courant **Français** : Courant **Anglais** : Scientifique

Centres d'intérêts

- La Cyber Security et les nouvelles technologies vis-à-vis des défis de sécurité.
- Recherche scientifique et innovation.
- Volontariat (Secouriste volontaire à la croix rouge Maroc 2011).
- Lecture, Voyage, Football.