Université Paris Cité

École Doctorale Sciences Mathématiques de Paris Centre (ED 386)
Institut de Recherche en Informatique Fondamentale

Thèse de Doctorat en Informatique

Specification and Verification of Isolation Levels in Distributed Storage Systems

présentée par

Enrique Román Calvo

dirigée par:

Directeur de thèse: Ahmed Bouajjani, co-Directeur: Constantin Enea

Soutenue publiquement le 19 Décembre 2025 devant un jury composé de:

Acknowledgments

Anyone in academia would agree that the acknowledgments of a thesis is the most read one. What follows is a (probably non-exhaustive) enumeration of disjoint sets of people that helped me and supported me in this trip (in multiple, non-disjoint ways).

First and foremost, I would like to thank my advisors, Ahmed Bouajjani and Constantin Enea, for all of their support. Working with you during these four years was a great pleasure, full of interesting problems and long nights before deadlines to solve them. Thanks to your support I think I slightly learned how to think one step beyond and I grew both as a person and as a researcher. Special thanks to Hagit Attiya, both for introducing me to the distributed systems world as well as for your positive support during our two-year-long collaboration.

I would like to thank my PhD defence jury members: Brijesh Dongol, Viktor Vafeiadis, Azadeh Farzan, Eric Koskinen and Anca Muscholl; and especially Brijesh Dongol and Viktor Vafeiadis for accepting to review my thesis and for giving me constructive feedback for improving it.

Being at IRIF would not have been the same without my officemates, from Mouna, Filippo, Weiqiang and Srinidhi NAGENDRA, with whom I spent amazing days and nights chatting, eating and partying, to the new recruits, Umberto, Cyril and Luc, who turned the office into a hotel room. Additional mentions to the officemates in the next office, i.e. Wael, Roman, Benjamin and Elie, as well as from even farther ones: Klara, Shamisa, Lucie, Clément, Arjan, Simona, Bernardo, Minh Hang, Victor, Vincent, Soumyajit, Dung, Avi, Robin, Salim and Sarah. Thank you for all those moments together! Special mention to my favorite climbing-gossip group, i.e. Mikaël, Daniel and Mónika. Wrapping up with IRIF, this trip would not have been possible without the support of the whole administrative team, especially without Omur and Jemuel.

During the last year, I had the opportunity to be a Pre-Post-Doc at the University of Freiburg under Andreas Podelski. I thank him and all his team for their warm welcome in Germany: Marcel, Ellie, Nico, Vincent, Lena, Tobias, Frank, Manuel, Abbie, Marlis, Martin, and especially to Dominik, with whom I spent and will spend a lot of time in verification and beyond.

Il faut également mentioner mes parisiens favoris : Ruth, Manu, Clément, Thomas, Anaïs, María et Quentin. Merci pour toutes les raclettes et les soirées à Sartrouville ! Et aussi Jorge, Francesco et Matija, avec qui j'ai découvert qu'on peut mélanger mille groupes d'amis en une seule nuit et ne jamais être trop nombreux.

No puedo no mencionar a mis 27 amigos de la uni, Rafa, Rubén, Guille, Cris, Ming, Chema, Alex (a.k.a. Cere) y Rivas (a.k.a. Pablo). Sin vosotros este viaje no hubiera ni empezado, gracias por todos los momentos en Madrid (y alrededores). También a mis eurovisivos, junto a quienes he disfrutado y sufrido largas noches de primavera: Álvaro, Luismi, Antonio, Noelia, Paula y Victor. Menzione speciale alle mie due italiane preferite, Shelly e Sara. Che ciò che Sky ha unito, Sanremo non lo separi.

Y aprovechamos que el Pisuerga pasa por Valladolid para mencionar a todos los fumbolistas, Iván, Javi, David, Diego García, Diego Aranda, Tania y Pablo. Y a mayores, mis matemáticos (y no tan matemáticos) vallisoletanos Isa, Jorge, Irene, Lucía, Antonio, Laura, Diego Sánchez (@Yo), Hervás, Jesús, Anita, Diego Munuera y Mia. ¡Nos vemos pronto!

Por supuesto, no podía faltar agradecer a toda mi familia por el apoyo durante todos estos años fuera de casa (y los que quedan): Mamá, Papá, Rubén, Marijose, Abuela, todos mis tíos y primos... ¡muchas gracias a todos!

And finally, thanks Dániel for belonging to each part of this whole trip of a PhD that we started togehter. Köszönöm szépen, ezek az évek nem lettek volna ugyanazok nélküled!

Last but not least, I would like to thank you for reading these acknowledgments, this thesis is also for you.

Abstract

Modern applications are centered around using large-scale distributed storage systems for storing and retrieving data. Storage systems provide different levels of consistency that correspond to different trade-offs between consistency and throughput. The weaker the consistency is, the more behaviors the storage is allowed to exhibit. In this thesis we address the problem of specifying and verifying a (distributed) storage system with respect to a given consistency model from three different perspectives.

The first problem we focus is studying the correctness of applications using database and isolation levels – database consistency models. We propose Stateless Model Checking algorithms that rely on Dynamic Partial Order Reduction. These algorithms work for a number of widely-used weak isolation levels, including Read Committed, Causal Consistency, Snapshot Isolation and Serializability. We show that they are complete, sound and optimal, and run with polynomial memory consumption in all cases. We report on an implementation of these algorithms in the context of Java Pathfinder applied to a number of challenging applications drawn from the literature of distributed databases.

The second question we focus on is the problem of testing isolation level implementations in databases, particularly when databases are accessed by transactions formed of SQL queries using multiple isolation levels at the same time. We show that many restrictions of this problem are NP-complete and provide an algorithm which is exponential-time in the worst-case, polynomial-time in relevant cases, and practically efficient.

The third problem we study is the fundamental tension between *availability* and *consistency* that shapes the design of distributed storage systems. Classical results capture extreme points of this trade-off: the CAP theorem shows that strong models like linearizability preclude availability under partitions, while weak models like causal consistency remain implementable without coordination. These theorems apply to simple read-write interfaces, leaving open a precise explanation of the combinations of object semantics and consistency models that admit available implementations. We develop a general semantic framework in which storage specifications combine operation semantics and consistency models. The framework encompasses a broad range of objects (key-value stores, counters, sets, CRDTs, and SQL databases) and consistency models (from causal consistency and sequential consistency to snapshot isolation and bounded staleness).

Within this framework, we prove the *Arbitration-Free Consistency* (AFC) theorem, showing that an object specification within a consistency model admits an available implementation if and only if it is *arbitration-free*, that is, it does not require a total arbitration order to resolve visibility or read dependencies. The AFC theorem unifies and generalizes previous results, revealing arbitration-freedom as the fundamental property that delineates coordination-free consistency from inherently synchronized behavior.

Keywords: Verification, Databases, Concurrency, Distributed Systems, Model-Checking, Testing, CAP Theorem, Isolation Levels, Partial Order Reduction

Résumé

Les applications modernes sont conçues autour de l'utilisation de systèmes de stockage distribués à grande échelle pour le stockage et la récupération des données. Les systèmes de stockage offrent différents niveaux de cohérence qui correspondent à différents compromis entre cohérence et débit. Plus la cohérence est faible, plus le stockage peut exhiber de comportements. Dans cette thèse, nous abordons le problème de spécifier et de vérifier un système de stockage (distribué) par rapport à un modèle de cohérence donné sous trois perspectives différentes.

Le premier problème sur lequel nous nous concentrons est l'étude de la correction des applications utilisant des bases de données et des niveaux d'isolation – modèles de cohérence des bases de données. Nous proposons des algorithmes de Vérification de Modèles sans État qui reposent sur la Réduction Dynamique par Ordre Partiel. Ces algorithmes fonctionnent pour un certain nombre de niveaux d'isolation faibles largement utilisés, y compris Read Committed, Causal Consistency, Snapshot Isolation et Serializability. Nous montrons qu'ils sont complets, corrects et optimaux, et qu'ils s'exécutent avec une consommation mémoire polynomiale dans tous les cas. Nous présentons une implémentation de ces algorithmes dans le cadre de Java Pathfinder, appliquée à plusieurs applications complexes issues de la littérature sur les bases de données distribuées.

La deuxième question que nous abordons concerne le problème du test des implémentations des niveaux d'isolation dans les bases de données, en particulier lorsque celles-ci sont accédées par des transactions composées de requêtes SQL utilisant plusieurs niveaux d'isolation simultanément. Nous montrons que de nombreuses variantes de ce problème sont NP-completes et proposons un algorithme dont la complexité est exponentielle dans le pire des cas, polynomiale dans les cas pertinents, et efficace en pratique.

Le troisième problème que nous étudions est la tension fondamentale entre *disponibilité* et *cohérence* qui façonne la conception des systèmes de stockage distribués. Les résultats classiques capturent les points extrêmes de ce compromis : le théorème CAP montre que des modèles forts comme la linéarisabilité excluent la disponibilité en cas de partitions, tandis que des modèles faibles comme la cohérence causale restent implémentables sans coordination. Ces théorèmes s'appliquent aux interfaces simples lecture-écriture, laissant ouverte une explication précise des combinaisons de sémantiques d'objets et de modèles de cohérence qui permettent des implémentations disponibles. Nous développons un cadre sémantique général dans lequel les spécifications de stockage combinent les sémantiques des opérations et les modèles de cohérence. Ce cadre englobe un large éventail d'objets (magasins clé-valeur, compteurs, ensembles, CRDTs et bases de données SQL) et de modèles de cohérence (de la cohérence causale et séquentielle à l'isolation par instantané et la staleness bornée).

Dans ce cadre, nous démontrons le théorème d'*Arbitration-Free Consistency* (AFC), montrant qu'une spécification d'objet au sein d'un modèle de cohérence admet une implémentation disponible si et seulement si elle est *sans arbitrage*, c'est-à-dire qu'elle ne nécessite pas d'ordre total d'arbitrage pour résoudre les dépendances de visibilité ou de lecture. Le théorème AFC unifie et généralise les résultats précédents, révélant que l'absence d'arbitrage est la propriété fondamentale qui distingue la cohérence sans coordination du comportement nécessitant une synchronisation intrinsèque.

Mots-clés: Vérification, Bases de données, Concurrence, Systèmes distribués, Vérification de modèles, Tests, Théorème CAP, Niveaux d'isolation, Réduction par ordre partiel

Contents

Résumé substantiel en français

Le stockage des données ne consiste plus à écrire des données sur un seul disque avec un unique point d'accès. Les applications modernes exigent non seulement la fiabilité des données, mais aussi des accès concurrents à haut débit. Par exemple, les applications liées aux chaînes d'approvisionnement, au secteur bancaire, etc., utilisent des bases de données relationnelles distribuées pour stocker et traiter les données, tandis que les applications comme les logiciels de réseaux sociaux et les plateformes de commerce en ligne utilisent des systèmes de stockage en nuage (tels qu'Azure Cosmos DB [92], Amazon DynamoDB [50], Facebook TAO [37]. . .).

Offrir un traitement à haut débit implique malheureusement un coût inévitable : l'affaiblissement des garanties de cohérence fournies aux utilisateurs. Pour illustrer ce phénomène, considérons une application bancaire où deux utilisateurs concurrents transfèrent de l'argent vers le même compte ; chacun lit le solde du compte et l'incrémente en conséquence. Pour garantir que les deux transferts aient lieu et que le solde total soit correctement mis à jour, l'un des expéditeurs doit attendre l'autre. Inversement, garantir uniquement des formes faibles de cohérence peut améliorer le débit, mais au prix de permettre que des clients connectés simultanément observent différentes versions d'une même donnée. En reprenant l'exemple, si seules des garanties faibles sont appliquées, les deux clients peuvent observer l'état initial, avant exécution, et mettre à jour le solde du compte uniquement en fonction de l'état qu'ils ont observé, sans remarquer l'autre transfert concurrent. Le résultat d'une telle exécution est que le solde du compte peut ne refléter qu'un seul des deux transferts, les utilisateurs pouvant potentiellement perdre de l'argent !

Dans le cas particulier des bases de données, ces "anomalies" peuvent être évitées en utilisant un *niveau d'isolation* fort, tel que la *Sérialisabilité* [90], qui offre essentiellement une version unique des données à tous les clients à tout moment. Cependant, la sérialisabilité nécessite une synchronisation coûteuse et entraîne un important surcoût en termes de performance. De nombreux systèmes de stockage modernes sacrifient la cohérence forte au profit de meilleures performances et garantissent des notions plus faibles d'isolation, telles que la *Transactional Causal Consistency* [75, 81, 12], la *Snapshot Isolation* [27], la *Read Committed* [27], etc., afin d'améliorer les performances. Dans une enquête récente menée auprès d'administrateurs de bases de données [91], 97% des participants ont indiqué que la plupart ou la totalité des transactions dans leurs bases de données s'exécutaient sous des niveaux d'isolation faibles.

Un niveau d'isolation plus faible permet davantage de comportements possibles que des modèles de cohérence plus forts. Il revient donc aux développeurs de s'assurer que leur application peut tolérer cet ensemble élargi de comportements. Malheureusement, les niveaux d'isolation faibles sont difficiles à comprendre et à raisonner [38, 9], et les bogues applicatifs qui en résultent peuvent entraîner des pertes commerciales [104].

Plusieurs travaux ont étudié le problème de la compréhension des niveaux d'isolation. La norme ANSI des niveaux d'isolation des transactions SQL a été introduite en 1992 [1]. Les niveaux ANSI décrivent de manière informelle les exigences minimales pour les niveaux d'isolation, permettant certains comportements dans les niveaux d'isolation faibles qui étaient interdits dans les niveaux plus forts. En 1995 [26], il a été critiqué que les niveaux ANSI définis à l'aide d'énoncés en anglais n'étaient pas adéquats, car ils étaient parfois imprécis, parfois ambiguës ; et ils ont proposé une nouvelle formalisation des niveaux d'isolation basée sur des mécanismes de verrouillage. Hélas, dans 1999, [9] a montré que l'intuition derrière la norme ANSI ne correspondait pas à la définition formelle proposée dans [26]. Au lieu de cela, Adya [9] a proposé une formalisation alternative en étudiant les graphes de dépendance entre transactions qui capturaient l'intuition de l'ANSI.

Dans 2017, le travail de Pavlo a révélé un manque de recherches sur l'impact du niveau d'isolation sur la correction [91]. Pavlo a soutenu qu'en général, les chercheurs se concentrent sur l'obtention de garanties fortes et bien comprises, telles que la Sérialisabilité, au lieu de garanties plus faibles mais largement utilisées, telles que la Read Committed. Les protocoles de contrôle de la concurrence utilisés dans les bases de données à grande échelle pour implémenter des niveaux d'isolation basés sur la formalisation proposée dans [9] sont difficiles à construire et à tester. Par exemple, le cadre de test boîte noire Jepsen [13] a trouvé un nombre remarquablement élevé de problèmes subtils dans de nombreuses bases de données en production. Plusieurs travaux ont abordé le problème de la vérification du code en production en présence de niveaux d'isolation en fournissant des outils pour détecter ou prévenir les bogues [29, 30, 69, 13].

Dans cette thèse, nous nous concentrons sur la vérification des programmes transactionnels et des bases de données en présence de niveaux d'isolation. En particulier, nous étudions comment effectuer une vérification par model checking des programmes transactionnels en supposant que la base de données fournit correctement un certain niveau d'isolation, ainsi que comment tester si la base de données fournit effectivement les niveaux d'isolation mentionnés en analysant la correction des exécutions transactionnelles SQL.

Au-delà de la vérification, nous nous interrogeons également sur les niveaux d'isolation pour lesquels il existe des implémentations permettant un haut débit, dans le but de concilier recherche et développement. Inspirés par des implémentations modernes de bases de données telles que CockroachDB [99] et TiDB [66], nous nous concentrons sur les implémentations distribuées et répliquées. Comme mentionné précédemment, la Sérialisabilité n'est pas un niveau d'isolation permettant un haut débit, car elle nécessite une synchronisation pour implémenter la Sérialisabilité [59]. En général, synchroniser des clients concurrents dans un scénario distribué empêche la disponibilité, que nous considérions des transactions ou non. Nous explorons donc un scénario plus général, où les transactions ne jouent plus un rôle prééminent et où les niveaux d'isolation sont englobés dans les modèles de cohérence. Dans ce cadre, nous étudions quels objets soutiennent les implémentations disponibles de quels modèles de cohérence.

Pour résumer, dans cette thèse, nous abordons les questions suivantes :

1. *Comment pouvons-nous effectuer un model checking efficacement pour des applications reposant sur des bases de données sous des niveaux d'isolation transactionnelle ?*

2. *Quelle est la complexité de la vérification de la cohérence d'une exécution SQL par rapport à une configuration d'isolation ?*

3. *Quelle classe de modèles de cohérence soutient les implémentations disponibles de quels objets en présence de partitions réseau ?*

Vérification des Applications Basées sur des Bases de Données

Le model checking [48, 94] est une technique de vérification efficace pour vérifier si un modèle à états finis satisfait une spécification donnée. Il explore l'espace d'états d'un programme de manière systématique et offre une large couverture du comportement du programme. Cependant, il se heurte au fameux problème d'explosion de l'état, c'est-à-dire que le nombre d'exécutions croît de manière exponentielle en fonction du nombre de clients concurrents.

La réduction de l'ordre partiel (Partial Order Reduction, POR) [49, 60, 93, 102] est une approche qui limite le nombre d'exécutions explorées sans sacrifier la couverture. La POR repose sur une relation d'équivalence entre les exécutions où, par exemple, deux exécutions sont équivalentes si l'une peut être obtenue à partir de l'autre en échangeant des étapes d'exécution consécutives indépendantes (non conflictuelles). Les techniques de POR garantissent qu'au moins une exécution de chaque classe d'équivalence est explorée. Les techniques de POR *optimales* explorent exactement une exécution de chaque classe d'équivalence. Au-delà de cette notion classique d'optimalité, les techniques de POR peuvent viser l'optimalité en évitant de visiter des états à partir desquels l'exploration est bloquée. La *réduction dynamique de l'ordre partiel* (Dynamic Partial Order Reduction, DPOR) [54] a été introduite pour explorer l'espace d'exécution (et suivre la relation d'équivalence entre les exécutions) à la volée, sans se baser sur des analyses statiques a priori. Cela est généralement couplé avec le *stateless model checking* (SMC) [61], qui explore les exécutions d'un programme sans stocker les états visités, évitant ainsi une consommation excessive de mémoire.

Ces dernières années, certains travaux ont étudié la DPOR dans le cas de programmes à mémoire partagée s'exécutant sous des modèles de mémoire faibles tels que TSO ou Release-Acquire, par exemple [4, 5, 7, 72]. Bien que ces algorithmes soient valides et complets, ils ont une complexité en espace exponentielle lorsqu'ils sont optimaux. Plus récemment, [73] a défini un algorithme DPOR qui a une complexité en espace polynomial, tout en étant valide, complet et optimal. Cet algorithme peut être appliqué à une gamme de modèles de mémoire partagée.

Bien que tous les travaux mentionnés concernent des programmes à mémoire partagée, nous ne connaissons aucune publication abordant le cas des programmes transactionnels de base de données avant notre contribution.

Dans cette thèse, nous présentons le premier algorithme DPOR SMC pour une base de données clé-valeur sous des niveaux d'isolation transactionnelle avec un ensemble statique de clés, qui est à la fois *valide*, c'est-à-dire qu'il énumère uniquement les exécutions *réalisables* par rapport au niveau d'isolation, *complet*, c'est-à-dire qu'il génère un représentant de chaque classe d'équivalence, *optimal*, c'est-à-dire qu'il génère *exactement une* exécution complète de chaque classe d'équivalence, et utilise *une mémoire polynomiale*. Il généralise l'approche adoptée par [72, 73], qui énumère les historiques des exécutions du programme. Nous avons étudié les limitations théoriques de notre approche et évalué notre algorithme sur un certain nombre d'applications complexes de bases de données tirées de la littérature.

Analyse de Complexité de la Vérification des Niveaux d'Isolement SQL

La formalisation des niveaux d'isolation SQL présentée par Adya dans [9] repose sur l'absence de dépendances cycliques dans des graphes de dépendances spécifiques. Dans [45], Cerone et al. ont critiqué certaines des définitions proposées comme étant de bas niveau, et a présenté une caractérisation alternative des niveaux d'isolation lorsque les transactions sont composées de lectures et d'écritures sur un ensemble *statique* de clés. Dans [29], Biswas and Enea ont proposé une définition alternative mais équivalente, adaptée à l'étude de la complexité de la vérification si une exécution donnée adhère aux sémantiques du niveau d'isolation prescrit, lorsque les transactions sont composées d'opérations de lecture et d'écriture sur une seule clé.

Dans cette thèse, nous considérons le problème de la vérification des implémentations des niveaux d'isolation dans les bases de données dans un cadre plus général que [29], inspiré par des scénarios qui se produisent dans les logiciels commerciaux [91]. Plus précisément, nous étudions la vérification si une exécution SQL est cohérente par rapport aux niveaux d'isolation prescrits, où les transactions sont formées de requêtes SQL et *plusieurs* niveaux d'isolation sont utilisés simultanément, c'est-à-dire que chaque transaction se voit attribuer un niveau d'isolation potentiellement différent. 32% des répondants à l'enquête de [91] ont signalé qu'ils utilisent de telles configurations "hétérogènes".

En tant que première contribution, nous introduisons une sémantique formelle axiomatique pour les exécutions avec des transactions SQL et une gamme de niveaux d'isolation, y compris la Sérialisabilité, la Snapshot Isolation, la Prefix Consistency, la Transactional Causal Consistency et la Read Committed. Traiter des requêtes SQL est plus complexe que les instructions classiques de lecture et d'écriture sur un ensemble *statique* de clés. En gros, les transactions SQL sont composées de quatre opérations : `INSERT`, `SELECT`, `UPDATE` et `DELETE`. Contrairement aux instructions de lecture et d'écriture statiques, `INSERT` et `DELETE` modifient l'ensemble des emplacements à l'exécution, tandis que l'ensemble des emplacements retournés ou modifiés par les requêtes `SELECT`, `UPDATE` et `DELETE` dépend de leurs valeurs (les valeurs sont restreintes pour satisfaire les clauses `WHERE`). Notre cadre générique pour définir les sémantiques des niveaux d'isolation adapte le travail de [29] et nous permet d'obtenir une sémantique équivalente à celle introduite par Adya dans [9].

Nous fournissons les premiers résultats concernant la complexité de la vérification de la correction des implémentations de niveaux d'isolation mixtes pour les transactions SQL. Nous montrons que prendre en compte les sémantiques de type SQL est strictement plus complexe que de simples lectures et écritures sur un ensemble *statique* de clés (NP-difficile dans la plupart des cas). En particulier, nos résultats montrent que la prise en compte des requêtes de suppression joue un rôle clé dans la divergence par rapport à l'analyse de complexité de [29]. Nous présentons également un premier outil qui peut être utilisé pour tester leur correction, et discutons de certaines conditions suffisantes sur les bancs d'essai pour lesquels les vérifications de cohérence peuvent être effectuées en temps polynomial.

Implémentations Disponibles des Systèmes de Stockage Distribués

Les systèmes de stockage distribués permettent un accès fiable aux objets en les répliquant à travers un réseau étendu. La réplication est essentielle pour tolérer les fautes dans le système (par exemple, les pannes de machines, les partitions réseau) et pour réduire la latence. Dans de tels systèmes, il est crucial de maintenir un compromis entre la disponibilité (garantir un accès rapide aux données) et la préservation de la cohérence, même en présence de délais de communication. Le *théorème CAP* [59, 36] montre qu'un magasin clé-valeur ne peut pas fournir une *Cohérence* forte (atomicité) tout en maintenant la *Disponibilité* (Availability) et en tolérant les *Partitions* du réseau simultanément. PACELC [3, 62] affine CAP en ajoutant le cas d'un réseau connecté où une cohérence forte ne peut pas être atteinte avec une faible latence.

De nombreux systèmes de stockage modernes sacrifient la cohérence forte au profit de la disponibilité (ou d'une faible latence) et garantissent des notions plus faibles de cohérence. Il existe une pléthore de modèles de cohérence faible [40] (ou *niveaux d'isolation* [10] dans le contexte des transactions) qui correspondent à différents compromis en matière de disponibilité. D'autres systèmes de stockage modernes assouplissent la sémantique des objets qu'ils prennent en charge, par exemple les registres à valeurs multiples, où une opération `get` renvoie arbitrairement une valeur précédemment stockée.

Les résultats précédents n'apportaient que des réponses partielles à cette question. Le théorème CAP mentionné ci-dessus ne fournit qu'un résultat négatif, à savoir qu'un *magasin clé-valeur atomique (linéarisable)* n'appartient pas à cette classe. Attiya et al. [19] identifient un modèle de cohérence, appelé Observable Causal Consistency (OCC), qui n'est pas inclus dans cette classe ; mais seulement pour certains objets particuliers, les registres à valeurs multiples. Nous remarquons que la cohérence causale, qui est strictement plus faible que les deux, appartient à la classe.

Dans cette thèse, nous apportons une réponse précise à la question soulevée ci-dessus. Pour ce faire, nous nous appuyons sur un cadre très expressif pour définir des modèles de cohérence et des sémantiques d'objets, fondé sur des travaux antérieurs [40]. En utilisant ce cadre, nous donnons une caractérisation *tight* des modèles et des objets qui peuvent être exprimés dans ce cadre et qui admettent des implémentations disponibles.

Notre résultat principal affirme, de manière approximative, qu'un système de stockage dispose d'une implémentation disponible si et seulement s'il met en œuvre un modèle de cohérence *sans arbitrage*, c'est-à-dire un modèle de cohérence dont les *formules de visibilité excluent toute utilisation significative de tout ordre total d'arbitrage*, c'est-à-dire de tout ordre total pouvant être utilisé comme “tie-breaker” pour fixer un ordre entre des invocations concurrentes.

1 | Introduction

Data storage is no longer about writing data to a single disk with a single point of access. Modern applications require not just data reliability, but also high-throughput concurrent accesses. For example, applications concerning supply chains, banking, etc. use distributed relational databases for storing and processing data, whereas applications such as social networking software and e-commerce platforms use cloud-based storage systems (such as Azure Cosmos DB [92], Amazon DynamoDB [50], Facebook TAO [37]. . .).

Providing high-throughput processing, unfortunately, comes at an unavoidable cost of weakening the consistency guarantees offered to users. To illustrate this phenomenon, let us consider a banking application where two concurrent users transfer money to the same account; reading the account balance and incrementing it accordingly. For ensuring both transfer take place and the total balance is correctly updated, one of the senders must wait for the other. Conversely, ensuring weak consistency guarantees can improve throughput, at the cost of allowing that concurrently-connected clients end up observing different versions of the same data. Following the example, if only weak guarantees are enforced, both clients may observe the initial state, prior execution, and update the account balance based only on their observed state without noticing the other concurrent transfer. The result of such execution is that the account balance may only reflect one of the two transfers, with users potentially losing money!

In the particular case of databases, these "anomalies" can be prevented by using a strong *isolation level* such as *Serializability* [90], which essentially offers a single version of the data to all clients at any point in time. However, serializability requires expensive synchronization and incurs a high performance cost. Many modern storage systems sacrifice strong consistency for better performance and ensure weaker notions of isolation, such as *Transactional Causal Consistency* [75, 81, 12], *Snapshot Isolation* [27], *Read Committed* [27], etc. for better performance. In a recent survey of database administrators [91], 97% of the participants responded that most or all of the transactions in their databases execute under weak isolation levels.

A weaker isolation level allows more possible behaviors than stronger consistency models. It is up to the developers to ensure that their application can tolerate this larger set of behaviors. Unfortunately, weak isolation levels are hard to understand or reason about [38, 9] and resulting application bugs can cause loss of business [104].

Multiple works investigated the problem of understanding isolation levels. The ANSI standard of SQL transaction isolation levels was introduced in 1992 [1]. The ANSI levels informally described minimal requirements for isolation levels, allowing some behaviors in weak isolation levels that were forbidden on stronger ones. In 1995 [26] criticized that the ANSI levels defined using English statements were not adequate, as they were sometimes imprecise, sometimes ambiguous; and they proposed a new formalization of isolation levels based on locking mechanisms. Alas, in 1999, [9] showed that the intuition behind the ANSI standard did not correspond to the formal definition proposed in [26]. Instead, Adya [9] proposed an alternative formalization studying dependency graphs between transactions that captured the ANSI intuition.

In 2017, the work by Pavlo exposed a lack of research on the impact of the isolation level on correctness [91]. Pavlo argued that in general, researchers focus on obtaining strong, well-understood guarantees, such as Serializability, instead of weaker but widely-employed ones such as Read Committed. The concurrency control protocols used in large-scale databases to implement isolation levels based on the formalization proposed in [9] are difficult to build and test. For instance, the black-box testing framework Jepsen [13] found a remarkably large number of subtle problems in many production databases. Multiple works approached the problem of verifying production code in the presence of isolation levels by providing tools for finding or preventing bugs [29, 30, 69, 13].

In this thesis, we focus on verification of both transactional programs and databases in the presence of isolation levels. In particular, we study how to model-check transactional programs assuming that the database correctly provides some isolation level, as well as how to test if the database indeed provides the aforementioned isolation levels analyzing the correctness of transactional SQL executions.

Beyond verification, we also question for which isolation levels there exist implementations that allow high-throughput; aiming to reconcile both research and development. Inspired by modern database implementations such as CockroachDB [99] and TiDB [66], we focus on distributed and replicated implementations. As mentioned before, Serializability is not an isolation level that allow high-throughput, as it requires synchronization for implementing Serializability [59]. In general, synchronizing concurrent clients on a distributed scenario disallows availability, whether we consider transactions or not. We thus explore a more general scenario, where transactions do not longer play a preeminent role anymore and isolation levels are subsumed into consistency models. There, we study which objects support available implementations of which consistency models.

To summarize, in this thesis we address the following questions:

1. *How can we efficiently model check database-backed applications under transaction isolation levels?*

2. *What is the complexity of checking if a SQL database execution is consistent with respect to an isolation configuration?*

3. *What class of consistency models support available implementations of which objects in the presence of network partitions?*

Model Checking Database-Backed Applications

Model checking [48, 94] is an effective verification technique for checking whether a finite-state model satisfies a given specification. It explores the state space of a given program in a systematic manner and it provides high coverage of program behavior. However, it faces the infamous state explosion problem, i.e., the number of executions grows exponentially in the number of concurrent clients.

Partial order reduction (POR) [49, 60, 93, 102] is an approach that limits the number of explored executions without sacrificing coverage. POR relies on an equivalence relation between executions where e.g., two executions are equivalent if one can be obtained from the other by swapping consecutive independent (non-conflicting) execution steps. POR techniques guarantee that at least one execution from each equivalence class is explored. *Optimal* POR techniques explore exactly one execution from each equivalence class. Beyond this classic notion of optimality, POR techniques may aim for optimality by avoiding visiting states from which the exploration is blocked. *Dynamic* partial order reduction (DPOR) [54] has been introduced to explore the execution space (and tracking the equivalence relation between executions) on-the-fly without relying on a-priori static analyses. This is typically coupled with *stateless* model checking (SMC) [61] which explores executions of a program without storing visited states, thereby, avoiding excessive memory consumption.

In the last few years, some works have studied DPOR in the case of shared memory programs running under weak memory models such as TSO or Release-Acquire, e.g. [4, 5, 7, 72]. While these algorithms are sound and complete, they have exponential space complexity when they are optimal. More recently, [73] defined a DPOR algorithm that has a polynomial space complexity, in addition of being sound, complete and optimal. This algorithm can be applied for a range of shared memory models.

While all the aforementioned works concern shared memory programs, we are not aware of any published work addressing the case of database transactional programs prior our contribution.

In this thesis, we present the first DPOR SMC algorithm for a key-value database under transaction isolation levels with a static set of keys that is at the same time *sound*, i.e., it enumerates only *feasible* executions with respect the isolation level, *complete*, i.e., it outputs a representative of each equivalence class, *optimal*, i.e., it outputs *exactly one* complete execution from each equivalence classm, and employs *polynomial memory.* It generalizes the approach adopted by [72, 73], which enumerates histories of program executions. We studied the theoretical limitations of our approach, and we evaluated our algorithm on a number of challenging database-backed applications drawn from the literature.

Complexity Analysis of Testing SQL Isolation Levels

The formalization of SQL isolation levels presented by Adya in [9] rely on the absence cyclic dependencies on specific dependency graphs. In [45], Cerone et al. criticized that some of the proposed definitions are low-level, and presented an alternative characterization of isolation levels when transactions are formed of reads and writes on a *static* set of keys. In [29], Biswas and Enea proposed an alternative yet equivalent definition tailored for studying the complexity of checking whether a given execution adheres to the prescribed isolation level semantics when transactions are composed of read and write operations on a single key.

In this thesis, we consider the problem of testing the isolation level implementations in databases in a more general setting than [29], inspired by scenarios that arise in commercial software [91]. More precisely, we study checking if a SQL execution is consistent with respect to the prescribed isolation levels, where transactions are formed of SQL queries and *multiple* isolation levels are used at the same time, i.e., each transaction is assigned a possibly different isolation level. 32% of the respondents of the survey in [91] signaled that they use such "heterogeneous" configurations.

As a first contribution, we introduce an axiomatic formal semantics for executions with SQL transactions and a range of isolation levels, including Serializability, Snapshot Isolation, Prefix Consistency, Transactional Causal Consistency and Read Committed. Dealing with SQL queries is more challenging than static read and write instructions. Roughly, SQL transactions are composed of four operations: `INSERT`, `SELECT`, `UPDATE` and `DELETE`. Unlike static read and write instructions, `INSERT` and `DELETE` change the set of locations at runtime, while the set of locations returned or modified by `SELECT`, `UPDATE` and `DELETE` queries depends on their values (the values are restricted to satisfy `WHERE` clauses). Our generic framework for defining isolation level semantics adapts the work of [29] and allow us to obtain an equivalent semantics to the aforementioned semantics introduced by Adya in [9].

We provide the first results concerning the complexity of checking the correctness of mixed isolation level implementations for SQL transactions. We show that considering SQL-like semantics is strictly more complex that just reads and writes on a *static* set of keys (NP-hard in most cases). In particular, our results show that considering delete queries plays a key role on the divergence with respect to the complexity analysis from [29]. We also present a first tool that can be used in testing their correctness, and discuss some sufficient conditions on benchmarks for which consistency checks can be done in polynomial time.

Available Implementations of Distributed Storage Systems

Distributed storage systems enable reliable access to objects by replicating them across a wide-area network. Replication is essential for tolerating faults in the system (e.g., machines that crash, network partitions) and for decreasing latency. In such systems, it is crucial to maintain a trade-off between availability (ensuring prompt access to data) and preserving consistency, even in the presence of communication delays. The *CAP theorem* [59, 36] shows that a key-value store cannot provide strong *Consistency* (atomicity) while maintaining *Availability* and tolerating network *Partitions* at the same time. PACELC [3, 62] refines CAP by adding the case of a connected network where strong consistency cannot be achieved with low latency.

Many modern storage systems sacrifice strong consistency for availability (or low latency) and ensure weaker notions of consistency. There is plethora of weak consistency models [40] (or *isolation levels* [10] in the context of transactions) that correspond to different trade-offs with respect to availability. Other modern storage systems relax the semantics of the objects they support, e.g., multi-value registers, where a get arbitrarily returns a previously stored value.

Given that the guarantees of a storage system are captured through the subtle combination of its consistency model and its object semantics, a natural question to consider is which class of consistency models support available implementations of which objects.

Previous results provided only partial answers to this question. The aforementioned CAP theorem only shows a negative result that an *Atomic (Linearizable) Key-Value Store* is not included in this class. Attiya et al. [19] identify a consistency model, called Observable Causal Consistency (OCC), that is not included in this class; but only for particular objects, Multi-Value Registers. We remark that Causal Consistency, which is strictly weaker than both of them, is in the class.

In this thesis we give a precise answer for the question raised above. To do so, we rely on a very expressive framework for defining consistency models and object semantics that builds on previous work [40]. Using this framework, we give a tight characterization of models and objects that can be expressed within this framework and that support available implementations.

Our main result states roughly, that a storage system has an available implementation if and only it implements an *arbitration-free* consistency model, i.e. a consistency models whose visibility formulas *exclude any meaningful use of any total arbitration order*, i.e. any total order that can be used as a "tie-breaker" to fix an order between concurrent invocations.

Structure

The rest of the thesis is organized as follows:

- Chapter 2 presents the notions of histories and isolation levels, central to the rest of the chapters.
- Chapter 3 introduces SMC algorithms and DPOR techniques for checking transaction isolation levels.
- Chapter 4 describes an axiomatic semantics of SQL transaction isolation and studies the complexity of checking if program executions are consistent with respect to them.
- Chapter 5 introduces the generic framework for describing consistency models and objects, and proves the AFC theorem.
- Chapter 6 summarizes our contributions and list open research topics.

Publications

This PhD thesis is based on the following works, where I am a primary author and a major contributor to their methodology, proofs, implementations and writing:

- Chapter 3: Dynamic Partial Order Reduction for Checking Correctness against Transaction Isolation Levels [32],
  Ahmed Bouajjani, Constantin Enea, Enrique Román-Calvo
  PLDI 2023
- Chapter 4: On the Complexity of Checking Mixed Isolation Levels for SQL Transactions [34],
  Ahmed Bouajjani, Constantin Enea, Enrique Román-Calvo
  CAV 2025
- Chapter 5: Arbitration-Free Consistency is Available (and Vice Versa) [20],
  Hagit Attiya, Constantin Enea, Enrique Román-Calvo
  POPL 2026

2 | Preliminaries

In this chapter we present an axiomatic formalization of databases that builds on the work of Biswas and Enea [29]. The extension to arbitrary storage is described in Chapter 5.

A *database* is formed of a set of infinite objects Objs, ranged over using x, y, z In the context of a relational database, objects correspond to fields/rows of a table while in the context of a key-value store, they correspond to keys. Client programs interact with the database via a non-empty set of operations (also called instructions) $\mathsf{InstrDB}$. An operation may read or write one or multiple objects; multi-object operations can be used to model SQL operations for example. Operations may write or return values in a set Vals. We assume that Vals contains a special value $\bot$, used for operations that do not return any value.

2.1 Transactions

We consider clients programs accessing the database from a number of parallel sessions, each session being a sequence of transactions defined by the following grammar:

$$\begin{aligned}
\mathsf{Transaction} &::= \mathtt{begin}; \mathsf{Body}; \mathtt{commit} \\
\mathsf{Body} &::= \mathsf{Instr} \ \mid\ \mathsf{Instr}; \mathsf{Body} \\
\mathsf{Instr} &::= \mathsf{InstrDB} \ \mid\ a := \mathsf{LExpr} \mid\ \mathtt{if}(\mathsf{LCond})\{\mathsf{Instr}\}
\end{aligned}$$

Each transaction is delimited by `begin` and `commit` instructions. The body contains statements for accessing the database and standard assignments and conditionals for local computation. Local computation uses local variables from a set LVars. We use a, b, ... to denote local variables. Expressions and Boolean conditions over local variables are denoted with LExpr and LCond, respectively. We leave unspecified the set of database access instructions ($\mathsf{InstrDB}$) as we consider in each chapter different ones, such as read-write operations (as in [29, 45]) or SQL-operations (as in [9]). We assume that $\mathsf{InstrDB}$ contains the instruction `abort`, allowing the client to abort the transaction and roll back all the modifications of the transaction.

The invocation of a database operation is represented using an *event*. We assume events are tuples $\langle \mathsf{id}, \mathsf{op} \rangle$, where id is an *event identifier* and op is a *type of operation*. For each event e, we denote by $\mathtt{id}(e)$ and $\mathtt{op}(e)$ to the type identifier, event identifier and type of an event respectively. Two distinct invocations of the same operation are considered different events, we represent this by associating each event to a unique event identifier. The set of events is denoted by Events. We assume that Events includes a special type of *initial events* that affect every object, representing an initial state of the database.

We assume that there is a type per database access statement, as well as two special types, `begin` and `commit`, indicating the start and the end of the transaction. We say that an event is of type `end` if it is either a `commit` or an `abort` event.

A *transaction log* (t, E, po_t) is an identifier t, a finite set of events E along with a strict total order po_t on E, called *program order* (representing the order between instructions in the body of a transaction). The set E of events in a transaction log t is denoted by $\mathsf{events}(t)$. For simplicity, we may use the term *transaction* instead of transaction log. Transactions always contain a single `begin` event; which is po_t-minimal w.r.t. po_t.

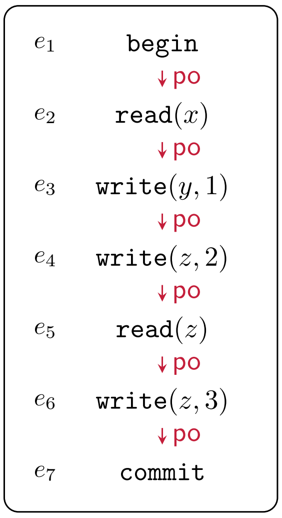


Figure 2.1: An example of a transaction using read and write semantics. Arrows represent po dependencies. We omit transitive edges and event identifiers for readability.

Figure 2.1 shows one example of transaction where the set of operations are reads and writes of a single object. A transaction may contain multiple reads and writes; and write multiple objects. In the following, we omit `begin` and `commit` events, as well as the program order relation in the transaction from our figures for readability.

A transaction with neither a `commit` nor an `abort` event is called *pending*. Otherwise, it is called *complete*. A complete transaction log with a `commit` event is called *committed* and *aborted* otherwise. If a `commit` or an `abort` event occurs, then it is maximal in po_t; `commit` and `abort` cannot occur in the same log. Note that a transaction is aborted because it executed an `abort` instruction. We do not consider transactions aborted by the database because their effect should not be visible to other transactions and the abort is not under the control of the program.

The event `read`(x) present in the transaction described in Figure 2.1 access the database while the event `write`$(y, 1)$ modifies it. In general, events associated to a operations reading the database are called *read* events. Similarly, events associated to operations writing the database are called *write* events. Events can be both read and write events, such as those whose operations are atomic read-writes, SQL updates, etc... Also, events such as `begin` or `commit` events are neither read nor write events.

For a given transaction t, we denote by $\mathsf{reads}(t)$ the set of read events contained in t. Also, if t does *not* contain an `abort` event, the set of write events in t is denoted by $\mathsf{writes}(t)$. If t contains an `abort` event, then we define $\mathsf{writes}(t)$ to be empty. This is because the effect of aborted transactions (its set of writes) should not be visible to other transactions. The extension to sets of transaction logs is defined as usual.

2.2 Histories

The interaction between a (client) program and a database is modeled as a *history* which records the operations executed on each session and data-flow dependencies that explain the values returned by operations on the database.

A *history* contains a set of transaction logs (with distinct identifiers) T ordered by a (partial) *session order* so that represents the order between transactions in the same session. It also includes a *write-read* relation wr (also known as read-from) representing data-flow dependencies between updates and reads; implicitly defining the values observed (returned) by the reads. For every key $x \in \mathsf{Keys}$ we consider a write-read relation $\mathsf{wr}_x \subseteq \mathsf{writes}(T) \times \mathsf{reads}(T)$. The union of wr_x for every $x \in \mathsf{Keys}$ is denoted by wr. We use h, h_1, h_2, ... to range over histories.

We assume that wr_x^{-1} is a (partial) function and thus, we use $\mathsf{wr}_x^{-1}(r)$ to denote the event w such that $(w, r) \in \mathsf{wr}_x$; if it is defined. Assuming wr_x^{-1} to be a function ensures that each read event r reads an object x from only one write event. Also, assuming that wr_x^{-1} is partial allows us defining histories where events do not read all possible objects, for example, if they are not present in the database because a previous event deleted them. We use $\mathsf{wr}_x^{-1}(r) \downarrow$ if there exists an event w such that $(w, e) \in \mathsf{wr}_x$, and $\mathsf{wr}_x^{-1}(r) \uparrow$, otherwise.

We say an event r *reads* an object x in a history if $\mathsf{wr}_x^{-1}(r) \downarrow$. In our model, we assume that if a transaction modifies an object x and then it reads the same object, then it must always return the value written in the transaction. Whenever a read r reads x from an event from the same transaction, we say that r *reads* x *locally* x, and we denote by $\mathsf{local}_x^{\mathsf{wr}}(r)$ to the write event $w = \mathsf{wr}_x^{-1}(r)$ that r reads x from. Otherwise, if r reads x but not locally, we say that r *reads* x *externally*, and we denote by $\mathsf{extern}_x^{\mathsf{wr}}(r)$ to the write event $w = \mathsf{wr}_x^{-1}(r)$ that r reads x from. We use $\mathsf{local}_x^{\mathsf{wr}}(r) \downarrow$ (resp. $\mathsf{extern}_x^{\mathsf{wr}}(r) \downarrow$) if r reads x locally (resp. externally), and $\mathsf{local}_x^{\mathsf{wr}}(r) \uparrow$ (resp. $\mathsf{extern}_x^{\mathsf{wr}}(r) \uparrow$) otherwise.

We represent the value written by a write event w on an object x via the *value function* $\mathtt{value}_{\mathsf{wr}}(w, x)$, a computable function that returns the value written by w (or $\perp$ if w does not write x). For example, if operations are reads and writes on a single object, as in Figure 2.1, the value written by an event is defined as follows:

$$\mathtt{value}_{\mathsf{wr}}(w, x) = \begin{cases} v & \text{if } w = \mathtt{write}(x, v) \\ \perp & \text{otherwise} \end{cases}$$

In general, we assume that $\mathtt{value}_{\mathsf{wr}}(w, x)$ is a computable function that returns the value written by w (or $\perp$ if w does not write x). The value written by an event w on an object x may depend on the value of x read by w. This is the case of atomic read-writes (also called conditional writes). We say that an event w *writes* x in h, denoted by w writes x, whenever $\mathtt{value}_{\mathsf{wr}}(w, x) \neq \perp$ and the transaction of w is *not* aborted.

We extend the relations wr and wr_x to pairs of transactions by $(t_1, t_2) \in \mathsf{wr}$, resp., $(t_1, t_2) \in \mathsf{wr}_x$, iff there exist events w in t_1 and r in $t_2, t_1 \neq t_2$ s.t. $(w, r) \in \mathsf{wr}$, resp., $(w, r) \in \mathsf{wr}_x$. Analogously, we extend wr and wr_x to tuples formed of a transaction (containing a write) and a read event. We say that the transaction t_1 is *read* by the transaction t_2 when $(t_1, t_2) \in \mathsf{wr}$. We assume that if a transaction contains multiple writes to the same object, then only the last one (w.r.t. po) can be read by other transactions; so the extension of wr to tuples formed of a transaction and a read event defines unambiguously a write-read relation of pairs of events. The extension of $\mathsf{extern}_x^{\mathsf{wr}}$ and $\mathsf{extern}^{\mathsf{wr}}$ is done similarly. We extend the function $\mathtt{value}$ to transactions: $\mathtt{value}_{\mathsf{wr}}(t, x)$ equals $\mathtt{value}_{\mathsf{wr}}(w, x)$, where w is the maximal event in po_t that writes x.

Histories contain an initial transaction, $\mathtt{init}$, that precedes every other transaction in T w.r.t so. We extend so to pairs of events by $(e_1, e_2) \in \mathsf{so}$ if $(\mathsf{tr}(e_1), \mathsf{tr}(e_2)) \in \mathsf{so}$. Also, $\mathsf{po} = \bigcup_{t \in T} \mathsf{po}_t$. The set of events present in h is denoted by $\mathsf{events}(h)$.

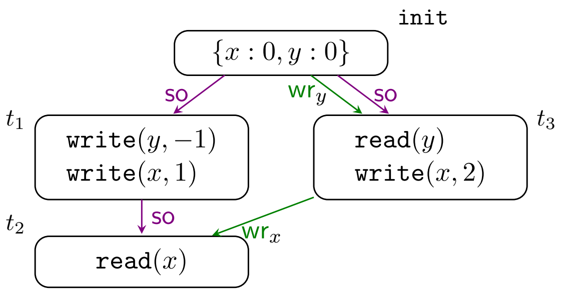


Figure 2.2: An example of a history where transactions use read and write operations. Arrows represent so and wr relations. The initial transaction $\mathtt{init}$ defines the initial state where both x and y are 0. Transaction t_1 writes on x and y value 1 and -1 respectively while transaction t_3 writes on x value 2 and 2 respectively. Transaction t_2 is executed on the same session as t_1, but reads x from transaction t_3; while transaction t_3 reads y from the initial transaction

Figure 2.2 presents a history in the context of [29]. Transactions omit $\mathtt{begin}$ and $\mathtt{commit}$ events for legibility.

Definition 2.2.1. *A* history $(T, \mathsf{so}, \mathsf{wr})$ *is a set of transactions T along with a strict partial* session order so, *and a* write-read *relation* $\mathsf{wr}_x \subseteq \mathsf{writes}(T) \times \mathsf{reads}(T)$ *for every* $x \in \mathsf{Objs}$ *s.t.*

- *T contains a single initial event $\mathtt{init}$ which is a so-predecessor of every other transaction,*
- *the inverse of wr_x is a function,*
- *if r reads x from w, then w writes x: if $\mathsf{wr}_x^{-1}(r) \downarrow$, $\mathtt{value}_{\mathsf{wr}}(\mathsf{wr}_x^{-1}(r), x) \neq \bot$,*
- *if r reads x externally, then r cannot read x locally: if $\mathsf{extern}_x^{\mathsf{wr}}(r) \downarrow$, then $\mathsf{local}_x^{\mathsf{wr}}(r) \uparrow$,*
- *if r reads x locally, then r reads x from the latest event preceding r in $\mathsf{tr}(r)$ that writes x: if $\mathsf{local}_x^{\mathsf{wr}}(r) \downarrow$, then W_x^r is not empty and $\mathsf{local}_x^{\mathsf{wr}}(r) = \max_{\mathsf{po}} W_x^r$, and*

- *$\mathsf{so} \cup \mathsf{wr}$ is acyclic (here we use the extension of wr to pairs of transactions).*

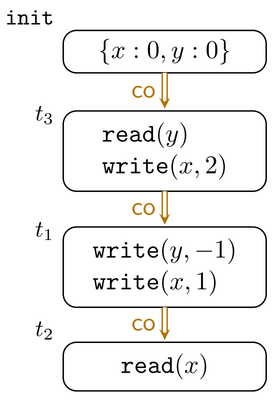
where $W_x^r = \{w \in \mathtt{tr}(r) \mid (w, r) \in \mathsf{po} \land \mathtt{value}_{\mathsf{wr}}(w, x) \neq \bot\}$.

2.3 Executions

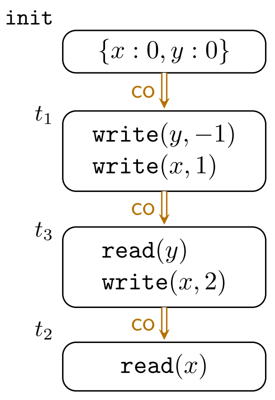
An *execution* on a database is represented abstractly using a history with a set of transactions T along with a well-founded total order $\mathsf{co} \subseteq T \times T$ called *commit order*. Intuitively, the commit order represents the order in which transactions are committed in the database.

Definition 2.3.1. *An* execution *$\xi = (h, \mathsf{co})$ is a history $h = (T, \mathsf{so}, \mathsf{wr})$ along with a commit order $\mathsf{co} \subseteq T \times T$, s.t. transactions in the same session or that are read are necessarily committed in the same order: $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{co}$. ξ is called an execution of h.*

For a transaction t, we use $t \in \xi$ to denote the fact that $t \in T$. Analogously, for an event e, we use $e \in \xi$ to denote that $e \in t$ and $t \in \xi$. The extension of a commit order to pairs of events or pairs of transactions and events is done in the obvious way.



(a) An execution execution of the history in Figure 2.2.



(b) Another execution execution of the history in Figure 2.2.

Figure 2.3: Two different executions of the history from Figure 2.2. Arrows represent co relations. We omit transitive edges and the relations so and wr from the corresponding histories for readability.

Figure 2.3 shows the two possible executions of the history $h = (T, \mathsf{so}, \mathsf{wr})$ depicted in Figure 2.2. As in any execution $\xi = (h, \mathsf{co})$ of h, $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{co}$, $\mathtt{init}$ must precede every transaction w.r.t. co, and transaction t_2 must succeed every transaction w.r.t. co.

2.4 Isolation Levels

Isolation levels enforce restrictions on the commit order in an execution that depend on the session order so and the write-read relation wr. An *isolation level* ι is a set of constraints called *axioms*. Intuitively, an axiom states that a read event r reads key x from transaction t_1 if t_1 is the latest transaction that writes x which is “visible” to r – latest refers to the commit order co. Formally, an axiom a is a predicate of the following form:

$$a(r) := \forall x, t_1, t_2 . t_1 \neq t_2 \land (t_1, r) \in \mathsf{wr}_x \land t_2 \text{ writes } x \land \mathsf{vis}_a(t_2, r, x) \implies (t_2, t_1) \in \mathsf{co} \tag{2.1}$$

where r is a read event from t.

The visibility relation of a vis_a is described by a formula of the form:

$$\mathsf{vis}_a(\tau_0, \tau_{k+1}, x) := \exists \tau_1, \dots, \tau_k . \bigwedge_{i=1}^{k+1} (\tau_{i-1}, \tau_i) \in \mathsf{Rel}_i \land \mathsf{wrCons}_a(\tau_0, \dots, \tau_{k+1}) \tag{2.2}$$

with each Rel_i is defined by the grammar:

$$\mathsf{Rel} ::= \mathsf{po} \mid \mathsf{so} \mid \mathsf{wr} \mid \mathsf{co} \mid \mathsf{Rel} \cup \mathsf{Rel} \mid \mathsf{Rel}; \mathsf{Rel} \mid \mathsf{Rel}^? \mid \mathsf{Rel}^+ \mid \mathsf{Rel}^* \tag{2.3}$$

This formula states that τ_0 (which is t_2 in Eq.2.1) is connected to τ_{k+1} (which is r in Eq.2.1) by a path of dependencies that go through some intermediate transactions or events $\tau_1, \dots, \tau_k$. Every relation used in such a path is described based on $\mathsf{po}, \mathsf{so}, \mathsf{wr}$ and co using union $\cup$, composition of relations ;, and transitive closure $^+$. $\mathsf{Rel}^?$ is syntactic sugar for $\mathtt{id} \cup \mathsf{Rel}$, and Rel^* for $\mathtt{id} \cup \mathsf{Rel}^+$; where $\mathtt{id}$ is the identity relation. Finally, extra requirements on the intermediate transactions s.t. writing a different key $y \neq x$ are encapsulated in the predicate $\mathsf{WrCons}_a(\tau_0, \dots, \tau_k, x)$.

Each axiom a uses a specific visibility relation denoted by vis_a. $\mathsf{vis}(\iota)$ denotes the set of visibility relations used in axioms defining an isolation level ι.

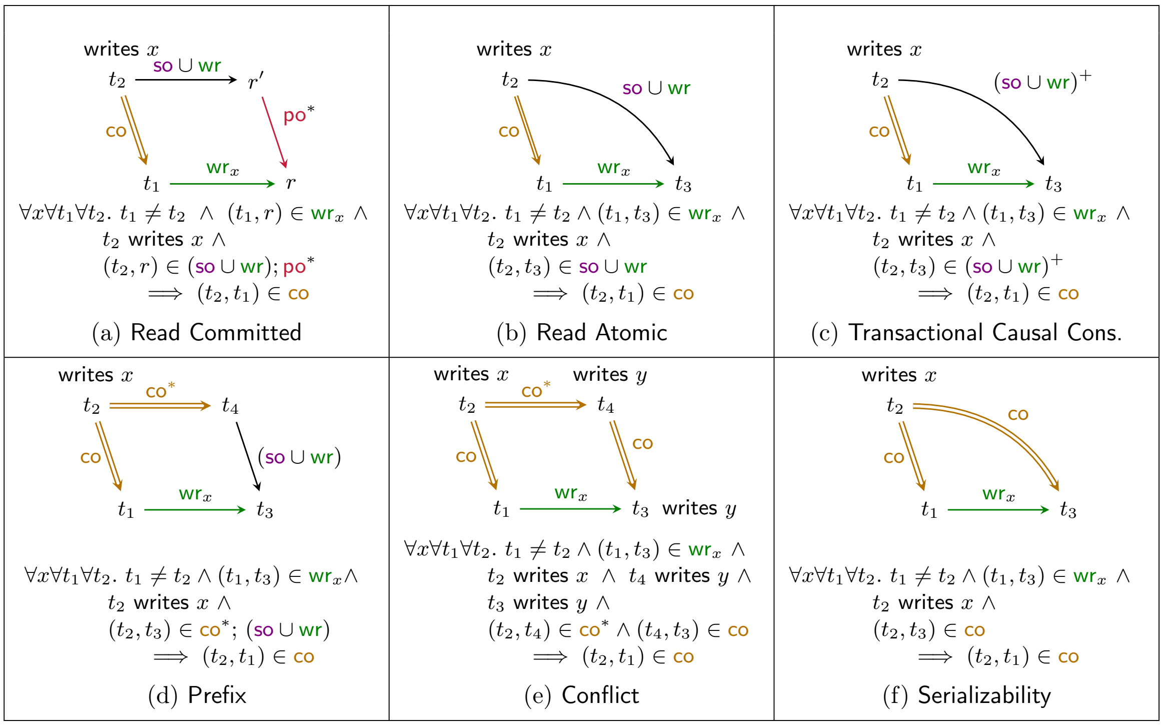
Figure 2.4 shows five axioms which correspond to their homonymous isolation levels [29]: *Read Committed* (RC), *Read Atomic* (RA), *Transactional Causal Consistency* (TCC), *Prefix Consistency* (PC) and *Serializability* (SER).

SER states that t_2 is visible to r if t_2 commits before r. Under PC, r observes a prefix of the transactions that precede $\mathtt{tr}(r)$ w.r.t. co. Such a prefix is indicated by the maximal transaction w.r.t. co t_4 that either precedes $\mathtt{tr}(r)$ w.r.t. so or that $\mathtt{tr}(r)$ reads from. TCC states that t_2 is visible to r if t_2 is in the causal past of t_3. RA states that t_2 is visible to r if either t_2 precedes $\mathtt{tr}(r)$ in its session or there exists an event r' in $\mathtt{tr}(r)$ that reads some object y from t_2. RC slightly varies from the requirements from RA as the event r' in $\mathtt{tr}(r)$ that reads from t_2 must precede r in the transaction.

In addition, the isolation level *Snapshot Isolation* (SI) is defined using two axioms: Prefix and Conflict. SI states that t_2 observes a prefix of the committed transactions and such prefix is conflict-maximal (i.e. if $\mathtt{tr}(r)$ writes some object y, the prefix it observes must contain all transactions writing y that precedes it w.r.t. co).

Note that SER is stronger than RC: every transaction visible to a read r according to RC is also visible to r according to SER. This means SER imposes more constraints for transaction t_1 to be read by r than RC.

In general, we say that an isolation level ι_1 is *stronger than* (respectively *weaker than*) another isolation level ι_2, denoted by $\iota_2 \preccurlyeq \iota_1$ (respectively $\iota_2 \succcurlyeq \iota_1$) if the number of constraints imposed for a transaction t_1 to be read by r imposed by ι_1 is larger than those imposed by ι_2. We observe that $\mathtt{RC} \preccurlyeq \mathtt{RA} \preccurlyeq \mathtt{TCC} \preccurlyeq \mathtt{PC} \preccurlyeq \mathtt{SI} \preccurlyeq \mathtt{SER}$.



$\forall x \forall t_1 \forall t_2.\ t_1 \neq t_2 \ \wedge\ (t_1, r) \in \mathsf{wr}_x \ \wedge$
t_2 writes x $\wedge$
$(t_2, r) \in (\mathsf{so} \cup \mathsf{wr}); \mathsf{po}^*$
$\implies (t_2, t_1) \in \mathsf{co}$

(a) Read Committed

$\forall x \forall t_1 \forall t_2.\ t_1 \neq t_2 \wedge (t_1, t_3) \in \mathsf{wr}_x \ \wedge$
t_2 writes x $\wedge$
$(t_2, t_3) \in \mathsf{so} \cup \mathsf{wr}$
$\implies (t_2, t_1) \in \mathsf{co}$

(b) Read Atomic

$\forall x \forall t_1 \forall t_2.\ t_1 \neq t_2 \wedge (t_1, t_3) \in \mathsf{wr}_x \ \wedge$
t_2 writes x $\wedge$
$(t_2, t_3) \in (\mathsf{so} \cup \mathsf{wr})^+$
$\implies (t_2, t_1) \in \mathsf{co}$

(c) Transactional Causal Cons.

$\forall x \forall t_1 \forall t_2.\ t_1 \neq t_2 \wedge (t_1, t_3) \in \mathsf{wr}_x \wedge$
t_2 writes x $\wedge$
$(t_2, t_3) \in \mathsf{co}^*; (\mathsf{so} \cup \mathsf{wr})$
$\implies (t_2, t_1) \in \mathsf{co}$

(d) Prefix

$\forall x \forall t_1 \forall t_2.\ t_1 \neq t_2 \wedge (t_1, t_3) \in \mathsf{wr}_x \ \wedge$
t_2 writes x $\wedge$ t_4 writes y $\wedge$
t_3 writes y $\wedge$
$(t_2, t_4) \in \mathsf{co}^* \wedge (t_4, t_3) \in \mathsf{co}$
$\implies (t_2, t_1) \in \mathsf{co}$

(e) Conflict

$\forall x \forall t_1 \forall t_2.\ t_1 \neq t_2 \wedge (t_1, t_3) \in \mathsf{wr}_x \ \wedge$
t_2 writes x $\wedge$
$(t_2, t_3) \in \mathsf{co}$
$\implies (t_2, t_1) \in \mathsf{co}$

(f) Serializability

Figure 2.4: Some well-known axioms defining isolations levels.

Given a history h and an isolation level ι, h is called *consistent* w.r.t. ι when there exists an execution ξ of h such that for all transactions t in ξ, the axioms in ι are satisfied in ξ (the interpretation of an axiom over an execution is defined as expected).

For example, let h be the history in Figure 2.2. The history h is not consistent w.r.t. `SER`: as discussed in Section 2.3, there are only two possible executions of h, ξ_1 (Figure 2.3a) and ξ_2 (Figure 2.3b). The execution ξ_1 is not consistent w.r.t. `SER` as t_2 reads x from t_3 but t_1 writes x and it is committed between t_3 and t_1. Also, the execution ξ_2 is not consistent w.r.t. `SER` as t_3 reads y from `init` but t_1 writes y and it is committed between `init` and t_3. The history h is consistent w.r.t. `TCC` as the execution ξ_2 does not violate the Transactional Causal Cons. axiom.

Definition 2.4.1. *A history $h = (T, \mathsf{so}, \mathsf{wr})$ is* consistent *with respect to the isolation level ι iff there is an execution ξ of h s.t. $\bigwedge_{t \in T, r \in \mathsf{reads}(t), a \in \iota} a(r)$ holds in ξ; ξ is called a* consistent *execution of h.*

3 Dynamic Partial Order Reduction for Checking Correctness against Transaction Isolation Levels

3.1 Introduction

In this chapter, we address the problem of *model checking* database-backed applications against a given isolation level. In particular, we focus on Dynamic Partial Order Reduction (DPOR) Stateless Model Checking (SMC) algorithms for databases with a static set of keys. We generalize the approach introduced by [73]. However, this generalization to the transactional case, covering the most relevant isolation levels, is not a straightforward adaptation of [73]. Ensuring optimality while preserving the other properties, e.g., completeness and polynomial memory complexity, is very challenging. Next, we explain the main steps and features of our work.

DPOR algorithms are parametrized by an equivalence relation on executions, most often, Mazurkiewicz equivalence [85]. In this work, we consider a weaker equivalence relation, also known as *read-from equivalence* [46, 7, 8, 72, 71, 73], which considers that two executions are equivalent when their histories are precisely the same (they contain the same set of events, and the relations po, so, and wr are the same). In general, reads-from equivalence is coarser than Mazurkiewicz equivalence, and its equivalence classes can be exponentially-smaller than Mazurkiewicz traces in certain cases [46].

Our SMC algorithms enumerate executions of a given program under a given isolation level ι. They are sound, complete and optimal. For isolation levels weaker than and including Transactional Causal Consistency, they satisfy a notion of *strong optimality* which says that additionally, the enumeration avoids states from which the execution is "blocked", i.e., it cannot be extended to a complete execution of the program. For Snapshot Isolation and Serializability, we show that *there exists* no algorithm in the same class (to be discussed below) that can ensure such a strong notion of optimality. All the algorithms that we propose are polynomial space, as opposed to many DPOR algorithms introduced in the literature.

As a starting point, we define a generic class of SMC algorithms, called *swapping based*, generalizing the approach adopted by [72, 73], which enumerate histories of program executions. These algorithms focus on the interaction with the database assuming that the other steps in a transaction concern local variables visible only within the scope of the enclosing session. Executions are extended according to a generic scheduler function Next and every read event produces several exploration branches, one for every write executed in the past that it can read from. Events in an execution can be swapped to produce new exploration "roots" that lead to different histories. Swapping events is required for completeness, to enumerate histories where a read r reads from a write w that is scheduled by Next after r. To ensure soundness, we restrict the definition of swapping so that it produces a history that is feasible by construction (extending an execution which is possibly infeasible may violate soundness). Such an algorithm is optimal w.r.t. the read-from equivalence when it enumerates each history exactly once.

We define a concrete algorithm in this class that in particular, satisfies the stronger notion of optimality mentioned above for every isolation level ι which is *prefix-closed* and *causally-extensible*, e.g., *Read Committed* and *Transactional Causal Consistency*. Prefix-closure means that every prefix of a history that satisfies ι, i.e., a subset of transactions and all their predecessors in the causal relation, i.e., $(\mathsf{so} \cup \mathsf{wr})^+$, is also consistent w.r.t. ι, and causal extensibility means that any pending transaction in a history that satisfies ι can be extended with one more event to still satisfy ι, and if this is a read event, then, it can read-from a transaction that precedes it in the causal relation. To ensure strong optimality, this algorithm uses a carefully chosen condition for restricting the application of event swaps, which makes the proof of completeness in particular, quite non-trivial.

We show that isolation levels such as Snapshot Isolation and Serializability are *not* causally-extensible and that there exists no swapping based SMC algorithm which is sound, complete, and strongly optimal at the same time (independent of memory consumption bounds). This impossibility proof uses a program to show that any Next scheduler and any restriction on swaps would violate either completeness or strong optimality. However, we define an extension of the previous algorithm which satisfies the weaker notion of optimality, while preserving soundness, completeness, and polynomial space complexity. This algorithm will simply enumerate executions according to a weaker prefix-closed and causally-extensible isolation level, and filter executions according to the stronger isolation levels Snapshot Isolation and Serializability at the end, before outputting.

We implemented these algorithms in the Java Pathfinder (JPF) model checker [103], and evaluated them on a number of challenging database-backed applications drawn from the literature of distributed systems and databases.

The rest of the chapter is structured as follows:

§ 3.2 recalls the specificities of the formalization of Biswas and Enea [29] with respect to generic framework presented in Chapter 2.

§ 3.3 identifies a class of isolation levels called prefix-closed and causally-extensible that admit efficient SMC.

§ 3.4 defines a generic class of swapping based SMC algorithms based on DPOR which are parametrized by a given isolation level.

§ 3.5 defines a swapping based SMC algorithm which is sound, complete, strongly-optimal, and polynomial space, for any isolation level that is prefix-closed and causally-extensible.

§ 3.6 shows that there exists no swapping based algorithm for Snapshot Isolation and Serializability, which is sound, complete, and strongly-optimal at the same time, and proposes a swapping based algorithm which satisfies "plain" optimality.

§ 3.7 reports on an implementation and evaluation of these algorithms.

§ 3.8 discusses related work.

§ 3.9 concludes with some remarks about our contribution.

3.2 Transactional Programs with Read-Write Operations

In this section we describe the concrete program syntax employed during the rest of the chapter.

3.2.1 Program Syntax

$$a \in \mathsf{LVars} \quad x \in \mathsf{Keys} \quad v \in \mathsf{Vals}$$

$$\begin{array}{llll} \mathsf{Trans} ::= \texttt{begin}; \mathsf{Body}; \texttt{commit} & & \mathsf{Instr} ::= \mathsf{InstrDB} \mid a := \mathsf{LExpr} \mid \texttt{if}(\mathsf{LCond})\{\mathsf{Instr}\} \\ \mathsf{Body} ::= \mathsf{Instr} \mid \mathsf{Instr}; \mathsf{Body} & & \mathsf{InstrDB} ::= a := \texttt{read}(x) \mid \texttt{write}(x, v) \mid \texttt{abort} \end{array}$$

Figure 3.1: Program syntax of a key-value store using read-write semantics.

Figure 3.1 lists the definition of a simple programming language that we use to represent applications running on top of a key-value database. We use Keys as an alias of Objs to indicate that our objects represent keys on the database; and we also call keys to objects.

Our results assume bounded programs, as usual in SMC algorithms, and therefore, we omit other constructs like `while` loops. SQL statements (`SELECT`, `JOIN`, `UPDATE`) manipulating relational tables can be compiled to reads or writes of keys representing rows in a table (see for instance, [95, 30]).

The instructions accessing the database correspond to reading the value of a key and storing it into a local variable a ($a := \texttt{read}(x)$), writing the value of a local variable a to a key x ($\texttt{write}(x, a)$), or aborting the transaction execution, rolling back all keys written by it. As described in Chapter 2, database instructions describe events with homonymous types. We call reads events to those whose type is `read`, and write events to those whose type is `write`. For a read/write event e, we use $\texttt{var}(e)$ to denote the key x.

3.2.2 Program Semantics

We define a small-step operational semantics for transactional programs, parametrized by an isolation level ι. The semantics keeps a history of previously executed database accesses in order to maintain consistency with ι.

For readability, we define a program as a partial function $\mathsf{P} : \mathsf{SessId} \rightharpoonup \mathsf{Sess}$ that associates session identifiers in SessId with concrete code as defined in Figure 3.1 (i.e., sequences of transactions). Similarly, the session order so in a history is defined as a partial function $\mathsf{so} : \mathsf{SessId} \rightharpoonup \mathsf{Tlogs}^*$ that associates session identifiers with sequences of transaction logs. Two transaction logs are ordered by so if one occurs before the other in some sequence $\mathsf{so}(j)$ with $j \in \mathsf{SessId}$.

The operational semantics is defined as a transition relation $\Rightarrow_I$ between *program configurations*, which are defined as tuples containing the following:

- history h storing the events generated by database accesses executed in the past,
- a valuation map $\vec{\gamma}$ that records local variable values in the current transaction of each session ($\vec{\gamma}$ associates identifiers of sessions with valuations of local variables),
- a map $\vec{B}$ that stores the code of each live transaction (mapping session identifiers to code),
- sessions/transactions P that remain to be executed from the original program.

The relation $\Rightarrow_I$ is defined using the set of rules described in Figure 3.2. Figure 3.2 uses the following notation. Let h be a history that contains a representation of so as above. We use $h \oplus_j (t, E, \mathsf{po}_t)$ to denote a history where (t, E, po_t) is appended to $\mathsf{so}(j)$. In particular, we use $h \oplus_j (e, \mathtt{begin})$ to denote the history where $(t, \{(e, \mathtt{begin})\}, \emptyset)$ with t a fresh id is appended to $\mathsf{so}(j)$. Also, for an event e, $h \oplus_j e$ is the history obtained from h by adding e to the last transaction log in $\mathsf{so}(j)$ and as a last event in the program order of this log (i.e., if $\mathsf{so}(j) = \sigma; (t, E, \mathsf{po}_t)$, then the session order so' of $h \oplus_j e$ is defined by $\mathsf{so}'(k) = \mathsf{so}(k)$ for all $k \neq j$ and $\mathsf{so}'(j) = \sigma; (t, E \cup \{e\}, \mathsf{po}_t \cup \{(e', e) : e' \in E\})$). Finally, for a history $h = (T, \mathsf{so}, \mathsf{wr})$, $h \oplus \mathsf{wr}(t, e)$ is the history obtained from h by adding (t, e) to the write-read relation.

We briefly describe the rules in Figure 3.2. The SPAWN rule starts a new transaction in a session j provided that this session has no other live transaction ($\vec{\mathsf{B}}(j) = \epsilon$). It adds a transaction log with a single `begin` event to the history and schedules the body of the transaction. IF-TRUE and IF-FALSE rules check the truth value of a Boolean condition of an `if` conditional. LOCAL models the execution of an assignment to a local variable which does not impact the stored history. READ-LOCAL and READ-EXTERN concern read instructions. READ-LOCAL handles the case where the read follows a write on the variable x in the same transaction: the read returns the value written by the last write on x in that transaction. READ-EXTERN corresponds to reading a value written in another transaction t'. The transaction t' is chosen non-deterministically as long as extending the current history with the write-read dependency associated to this choice leads to a history that still satisfies ι. Depending on the isolation level, there may not exist a transaction t' the read can read from (see Figure 3.13 for a concrete example). READ-EXTERN applies only when the executing transaction contains no write on the same variable. COMMIT confirms the end of a transaction making its writes visible while ABORT ends the transaction's execution immediately.

An *initial* program configuration for a program P contains the program P, a history $h = (\{t_0\}, \emptyset, \emptyset)$ where t_0 is a transaction log containing writes that write the initial value for

SPAWN
$$\frac{t \text{ fresh} \quad e \text{ fresh} \quad \mathsf{P}(j) = \texttt{begin}; \mathsf{Body}; \texttt{commit}; \mathsf{S} \quad \vec{\mathsf{B}}(j) = \epsilon}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h \oplus_j (t, \{(e, \texttt{begin})\}, \emptyset), \vec{\gamma}[j \mapsto \emptyset], \vec{\mathsf{B}}[j \mapsto \mathsf{Body}; \texttt{commit}], \mathsf{P}[j \mapsto \mathsf{S}]}$$

IF-TRUE
$$\frac{\psi(\vec{x})[x \mapsto \vec{\gamma}(j)(x) : x \in \vec{x}] \quad \vec{\mathsf{B}}(j) = \texttt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{Instr}; \mathsf{B}], \mathsf{P}}$$

IF-FALSE
$$\frac{\neg\psi(\vec{x})[x \mapsto \vec{\gamma}(j)(x) : x \in \vec{x}] \qquad \vec{\mathsf{B}}(j) = \texttt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

LOCAL
$$\frac{v = \vec{\gamma}(j)(e) \qquad \vec{\mathsf{B}}(j) = a := e; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h, \vec{\gamma}[(j,x) \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

WRITE
$$\frac{v = \vec{\gamma}(j)(x) \quad e \text{ fresh} \quad \vec{\mathsf{B}}(j) = \texttt{write}(x, x); \mathsf{B} \quad h \oplus_j (e, \texttt{write}(x, v)) \text{ satisfies } \iota}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h \oplus_j (e, \texttt{write}(x, v)), \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

READ-LOCAL
$$\frac{\mathsf{writes}(\texttt{last}(h, j)) \text{ contains a } \mathsf{write}(x, v) \text{ event} \qquad e \text{ fresh} \qquad \vec{\mathsf{B}}(j) = x := \texttt{read}(x); \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h \oplus_j (e, \texttt{read}(x)), \vec{\gamma}[(j,x) \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

READ-EXTERN
$$\frac{\begin{array}{c}\mathsf{writes}(\texttt{last}(h, j)) \text{ does not contain a } \mathsf{write}(x, v) \text{ event} \qquad e \text{ fresh} \qquad \vec{\mathsf{B}}(j) = x := \texttt{read}(x); \mathsf{B} \\ h = (T, \mathsf{so}, \mathsf{wr}) \qquad t = \texttt{last}(h, j) \qquad \mathsf{write}(x, v) \in \mathsf{writes}(t') \text{ with } t' \in \mathsf{cmtt}(h) \text{ and } t \neq t' \\ h' = (h \oplus_j (e, \texttt{read}(x))) \oplus \mathsf{wr}(t', e) \qquad h' \text{ satisfies } \iota\end{array}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h', \vec{\gamma}[(j,x) \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

COMMIT
$$\frac{e \text{ fresh} \quad \vec{\mathsf{B}}(j) = \texttt{commit}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h \oplus_j (e, \texttt{commit}), \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \epsilon], \mathsf{P}}$$

ABORT
$$\frac{e \text{ fresh} \quad \vec{\mathsf{B}}(j) = \texttt{abort}; B}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow_\iota h \oplus_j (e, \texttt{abort}), \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \epsilon], \mathsf{P}}$$

Figure 3.2: An operational semantics for transactional programs. Above, $\texttt{last}(h, j)$ denotes the last transaction log in the session order $\mathsf{so}(j)$ of h, and $\mathsf{cmtt}(h)$ denotes the set of transaction logs in h that are committed

.

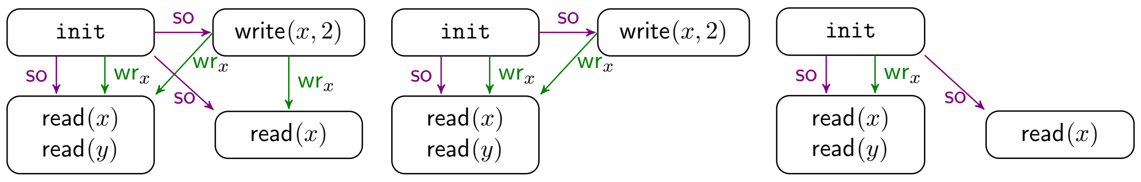
all variables, and empty current transaction code ($\mathsf{B} = \epsilon$). A program execution of a program P under an isolation level ι is a sequence of program configurations $c_0 \ldots c_n$ where c_0 is an initial configuration for P, and for every $0 \leq m < n$, $c_m \Rightarrow_I c_{m+1}$. We say that c_n is *reachable w.r.t.* ι from c_0. The history of such an execution is the history h in the last configuration c_n. A configuration is called *final* if it contains the empty program ($\mathsf{P} = \emptyset$). We denote by $\mathsf{hist}_\iota(\mathsf{P})$ to the set of all histories of an execution of P under ι that ends in a final configuration.

3.3 Prefix-Closed and Causally-Extensible Isolation Levels

We define two properties of isolation levels, prefix-closure and causal extensibility, which enable efficient DPOR algorithms (as shown in Section 3.5).

3.3.1 Prefix Closedness

For a relation $R \subseteq A \times A$, the restriction of R to $A' \times A'$, denoted by $R \downarrow A' \times A'$, is defined by $\{(a, b) : (a, b) \in R, a, b \in A'\}$. Also, a set A' is called R-downward closed when it contains $a \in A$ every time it contains some $b \in A$ with $(a, b) \in R$.



(a) A history. (b) A prefix. (c) Not a prefix.

Figure 3.3: Explaining the notion of prefix of a history. `init` denotes the transaction log writing initial values. Boxes group events from the same transaction.

A *prefix* of a transaction log (t, E, po_t) is a transaction log $(t, E', \mathsf{po}_t \downarrow E' \times E')$ such that E' is po_t-downward closed. A *prefix* of a history $h = (T, \mathsf{so}, \mathsf{wr})$ is a history $h' = (T', \mathsf{so} \downarrow T' \times T', \mathsf{wr} \downarrow T' \times T')$ such that every transaction log in T' is a prefix of a different transaction log in T but carrying the same id, $\mathsf{events}(h') \subseteq \mathsf{events}(h)$, and $\mathsf{events}(h')$ is $(\mathsf{po} \cup \mathsf{so} \cup \mathsf{wr})^*$-downward closed. For example, the history pictured in Fig. 3.3b is a prefix of the one in Fig. 3.3a while the history in Fig. 3.3c is not. The transactions on the bottom of Fig. 3.3c have a wr predecessor in Fig. 3.3a which is not included.

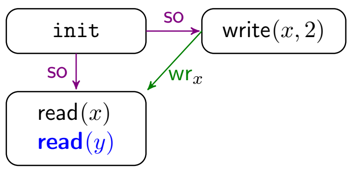
Definition 3.3.1. *An isolation level ι is called* prefix-closed *when every prefix of an consistent w.r.t. ι history is also consistent w.r.t. ι.*

Every isolation level ι discussed in Figure 2.4 is prefix-closed because if a history h is consistent w.r.t. ι with a commit order co, then the restriction of co to the transactions from a prefix h' of h satisfies the corresponding axiom(s) when interpreted over h'.

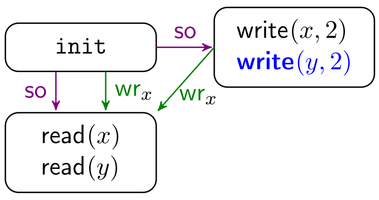
Theorem 3.3.2. *Read Committed, Read Atomic, Transactional Causal Consistency, Snapshot Isolation, and Serializability are prefix-closed.*

3.3.2 Causal Extensibility

We start with an example to explain causal extensibility. Let us consider the histories h_1 and h_2 in Figures 3.4a and 3.4b, respectively, *without* the events $\mathtt{read}(y)$ and $\mathtt{write}(y, 2)$ written in blue bold font. These histories satisfy Read Atomic. The history h_1 can be extended by adding the event $\mathsf{read}(y)$ and the wr dependency $\mathsf{wr}(\mathtt{init}, \mathsf{read}(y))$ while still satisfying Read Atomic. On the other hand, the history h_2 *can not* be extended with the event $\mathsf{write}(y, 2)$ while still satisfying Read Atomic. Intuitively, if the reading transaction on the bottom reads x from the transaction on the right, then it should read y from the same transaction because



(a) Extensible history.



(b) Non-extensible history.

Figure 3.4: Explaining causal extensibility. `init` denotes the transaction log writing initial values. Boxes group events from the same transaction.

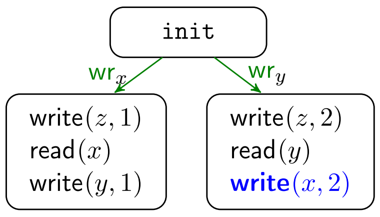


Figure 3.5: A counter-example to causal extensibility for `SI` and `SER`. The so-edges from `init` to the other transactions are omitted for legibility.

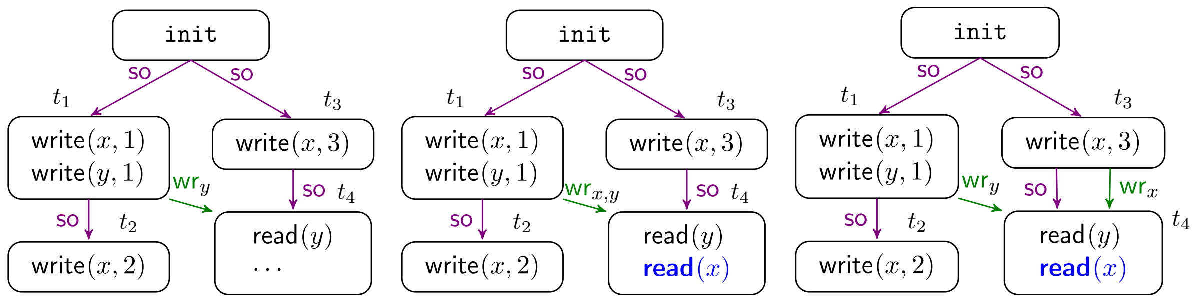
this is more "recent" than `init` w.r.t. session order. The essential difference between these two extensions is that the first concerns a transaction which is maximal in $(\mathsf{so} \cup \mathsf{wr})^+$ while the second no. The extension of h_2 concerns the transaction on the right in Figure 3.4b which is a wr predecessor of the reading transaction. Causal extensibility will require that at least the $(\mathsf{so} \cup \mathsf{wr})^+$ maximal (pending) transactions can always be extended with any event while still preserving consistency. The restriction to $(\mathsf{so} \cup \mathsf{wr})^+$ maximal transactions is intuitively related to the fact that transactions should not read from non-committed (pending) transactions, e.g., the reading transaction in h_2 should not read from the still pending transaction that writes x and later y.

Formally, let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history. A transaction t is called $(\mathsf{so} \cup \mathsf{wr})^+$-*maximal* in h if h does not contain any transaction t' such that $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^+$. We define a *causal extension* of a pending transaction t in h with an event e as a history h' such that:

- e is added to t as a maximal element of po_t,
- if e is a read event and t *does not* contain a write to $\mathtt{var}(e)$, then wr is extended with some tuple (t', e) such that $(t', t) \in (\mathsf{so} \cup \mathsf{wr})^+$ in h (if e is a read event and t *does* contain a write to $\mathtt{var}(e)$, then the value returned by e is the value written by the latest write on $\mathtt{var}(e)$ before e in t; the definition of the return value in this case is unique and does not involve wr dependencies),
- the other elements of h remain unchanged in h'.

For example, Figure 3.6b and 3.6c present two causal extensions with a $\mathtt{read}(x)$ event of the transaction t_4 in the history h in Figure 3.6a. The new read event reads from transaction t_1 or t_3 which were already related by $(\mathsf{so} \cup \mathsf{wr})^+$ to t_4. An extension of h where the new read event reads from t_2 is *not* a causal extension because $(t_2, t_4) \notin (\mathsf{so} \cup \mathsf{wr})^+$.

Definition 3.3.3. *An isolation level ι is called* causally-extensible *if for every consistent w.r.t. ι history h, every $(\mathsf{so} \cup \mathsf{wr})^+$-maximal pending transaction t in h, and every event e, there exists a causal extension h' of t with e that is consistent w.r.t. ι.*



(a) History h. (b) t_4 reads x and y from t_1. (c) t_4 reads x from t_3, y from t_1.

Figure 3.6: Two causal extensions of the history h on the left with the $\mathtt{read}(x)$ event written in blue.

Theorem 3.3.4. *Transactional Causal Consistency, Read Atomic, and Read Committed are causally-extensible.*

Proof. Let ι be an isolation level in $\{\mathtt{RA}, \mathtt{RC}, \mathtt{TCC}\}$ and let h be a history consistent w.r.t. ι. We show that for every $(\mathsf{so} \cup \mathsf{wr})^+$-maximal pending transaction t, there exists a causal extension of t in h, h', that is consistent w.r.t. ι. In particular, we show that if $\xi = (h, \mathsf{co})$ is a consistent execution of h w.r.t. ι, then $\xi' = (h', \mathsf{co})$ is also a consistent execution of h' w.r.t. ι. Two cases arise, depending on whether the event e is a read event or not.

On one hand, if e is not a read event, we consider $h' = h \oplus_j e$; where $j = \mathtt{ses}(t)$. Let us denote so' and wr' to the session order and write-read relations respectively used by h'. We reason by contradiction, assuming that ξ' is not consistent w.r.t. ι and reaching a contradiction. In such case, there must exists a variable x, a read event r and two distinct transactions t_1, t_2 s.t. $(t_1, r) \in \mathsf{wr}'_x$, $\mathsf{vis}_\iota(t_2, r, x)$ holds in ξ' but $(t_1, t_2) \in \mathsf{co}$; where vis_ι refers to the visibility relation of its homonymous axiom. We observe that, by construction of h', for every variable x and every event e, ${\mathsf{wr}'_x}^{-1}(e) = \mathsf{wr}_x^{-1}(e)$ and $\mathsf{so}' = \mathsf{so}$. Hence, by the definition of ι, $(t_1, r) \in \mathsf{wr}_x$ and $\mathsf{vis}_\iota(t_2, r, x)$ holds in ξ. However, this implies that ξ is not consistent w.r.t. ι; which is impossible by hypothesis. Therefore, we deduce that ξ' is consistent w.r.t. ι.

On the other hand, if e is a read event, we consider the history $h' \oplus_j e \oplus \mathsf{wr}(t_x, e)$; where $j = \mathtt{ses}(t)$, $x = \mathtt{var}(e)$ and t_x is the transaction described using Equation (3.1). Transaction t_x is well defined as $(\mathtt{init}, l_e) \in \mathsf{so}$. By the definition of $\mathtt{RA}, \mathtt{RC}$ and $\mathtt{TCC}$, h' is a causal extension of h.

$$t_x = \max_{\mathsf{co}} \{t \in T \mid \mathsf{vis}_\iota(t_2, l_e, x) \land t \text{ writes } x\} \tag{3.1}$$

where $l_e = \max_{\mathsf{po}} t$.

We prove the result once again by contrapositive, assuming that ξ' is not consistent w.r.t. ι. Let us denote by so' and wr' the session order and write-read relations of h'. In such case, there must exists a variable y, distinct transactions t_1, t_2 and read event r s.t. $(t_2, r) \in \mathsf{wr}'_y$, $\mathsf{vis}_\iota(t_2, r, y)$ holds in ξ' but $(t_1, t_2) \in \mathsf{co}$. First, for every event $e' \neq r$ and variable x, ${\mathsf{wr}'_x}^{-1}(e') = \mathsf{wr}_x^{-1}(r)$. Moreover, as t is $(\mathsf{so} \cup \mathsf{wr})^+$-maximal in h, it is also $(\mathsf{so}' \cup \mathsf{wr}')^+$-maximal in h'. Therefore, by the definitions of ι and t_x, if $e' \neq r$, $\mathsf{vis}_\iota(t_2, r, y)$ holds in ξ. Altogether, we deduce that if $e' \neq r$, $(t_2, r) \in \mathsf{wr}_y$, $\mathsf{vis}_\iota(t_2, r, y)$ holds in ξ and $(t_1, t_2) \in \mathsf{co}$. As ξ is consistent w.r.t. ι, this is not possible; so we deduce that $r = e$. If $r = e$, then $y = x$ and $t_1 = t_x$. However, by the definition of t_x, $(t_2, t_x) \in \mathsf{co}$; which is impossible as $(t_1, t_2) \in \mathsf{co}$ and co is a total order. In conclusion, the initial assumption is false; so co witnesses that h' is consistent w.r.t. ι. □

Snapshot Isolation and Serializability are *not* causally extensible. Figure 3.5 presents a counter-example to causal extensibility. Let h be the history that does *not* contain the $\texttt{write}(x, 2)$ written in blue bold font and let h' be the causal extension of h with this event. The history h' does not satisfy neither Snapshot Isolation nor Serializability although h does: the commit order co defined as $\texttt{init} <_{\mathsf{co}} < t_2 <_{\mathsf{co}} t_1$ is the only commit order witnessing h's consistency w.r.t. Snapshot Isolation and Serializability; but it does not witness consistency for h' as $(\texttt{init}, t_1) \in \mathsf{wr}_x$ but for every axiom $a \in \{\texttt{SER}, \texttt{SI}\}$, $\mathsf{vis}_a(t_2, t_1, x)$ holds using co. Note that the causal extension of a history with a write event is unique.

3.4 Swapping-Based Model Checking Algorithms

We define a class of stateless model checking algorithms for enumerating executions of a given transactional program, that we call *swapping-based algorithms*. Section 3.5 will describe a concrete instance that applies to isolation levels that are prefix-closed and causally extensible.

Algorithm 1 EXPLORE algorithm

```
1:  function EXPLORE($\mathsf{P}$, $h_<$, $\mathsf{locals}$)
2:    $j, e, \gamma \leftarrow$ NEXT($\mathsf{P}, h_<, \mathsf{locals}$)
3:    $\mathsf{locals}' \leftarrow \mathsf{locals}[e \mapsto \gamma]$
4:    if $e = \bot$ and VALID($h$) then
5:      output $h$, $\mathsf{locals}'$
6:    else if $\mathsf{op}(e) = \texttt{read}$ then
7:      for all $t \in$ VALIDWRITES($h, e$) do
8:        $h'_< \leftarrow h_< \oplus_j e \oplus \mathsf{wr}(t, e)$
9:        EXPLORE($\mathsf{P}, h'_<, \mathsf{locals}'$)
10:       EXPLORESWAPS($\mathsf{P}, h'_<, \mathsf{locals}'$)
11:   else
12:     $h'_< \leftarrow h_< \oplus_j e$
13:     EXPLORE($\mathsf{P}, h'_<, \mathsf{locals}'$)
14:     EXPLORESWAPS($\mathsf{P}, h'_<, \mathsf{locals}'$)
```

These algorithms are defined by the recursive function EXPLORE listed in Algorithm 1. The function EXPLORE receives as input a program P, an *ordered history* $h_<$, which is a pair $(h, <)$ of a history and a total order $<$ on all the events in h, and a mapping locals that associates each event e in h with the valuation of local variables in the transaction of e ($\mathsf{tr}(e)$)

just before executing e. For an ordered history $(h, <)$ with $h = (T, \mathsf{so}, \mathsf{wr})$, we assume that $<$ is consistent with po, so, and wr, i.e., $e_1 < e_2$ if $(\mathsf{tr}(e_1), \mathsf{tr}(e_2)) \in (\mathsf{so} \cup \mathsf{wr})^+$ or $(e_1, e_2) \in \mathsf{po}$. Initially, the ordered history and the mapping locals are empty.

The function EXPLORE starts by calling NEXT to obtain an event representing the next database access in some pending transaction of P, or a `begin`/`commit`/`abort` event for starting or ending a transaction. This event is associated to some session j. For example, a typical implementation of NEXT would be choosing one of the pending transactions (in some session j), executing all local instructions until the next database instruction in that transaction (applying the transition rules IF-TRUE, IF-FALSE and LOCAL), and returning the event e corresponding to such database instruction and the current local state γ. NEXT may also return $\perp$ if the program finished. If NEXT returns $\perp$, then the function VALID can be used to filter executions that satisfy the intended isolation level before outputting the current history and local states (the use of VALID will become relevant in Section 3.6).

Otherwise, the event e is added to the ordered history $h_<$. If e is a read event, then VALIDWRITES computes a set of write events w in the current history that are valid for e, i.e., adding the event e along with the wr dependency (w, e) leads to a history that still satisfies the intended isolation level.

Concerning notations, let $(h, <)$ be an ordered history where so is represented as a function $\mathsf{so} : \mathsf{SessId} \rightharpoonup \mathsf{Tlogs}^*$ (as in § 3.2.2). We define the ordered history $(h, <) \oplus_j e$ as $(h \oplus_j e, < \cdot e)$ where $< \cdot e$ means that e is added as the last element of $<$.

Algorithm 2 EXPLORESWAPS

```
1: function EXPLORESWAPS(P, $h_<$, locals)
2:     $l \leftarrow$ COMPUTEREORDERINGS($h_<$)
3:     for all $(\alpha, \beta) \in l$ do
4:         if OPTIMALITY($h_<, \alpha, \beta$, locals) then
5:             EXPLORE(P, SWAP($h_<, \alpha, \beta$, locals))
```

Once an event is added to the current history, the algorithm may explore other histories obtained by re-ordering events in the current one. Such re-orderings are required for completeness. New read events can only read from writes executed in the past which limits the set of explored histories to the scheduling imposed by NEXT. Without re-orderings, writes scheduled later by NEXT cannot be read by read events executed in the past, although this may be permitted by the isolation level.

The function EXPLORESWAPS calls COMPUTEREORDERINGS to compute pairs of sequences of events α, β that should be re-ordered; α and β are *contiguous and disjoint* subsequences of the total order $<$, and α should end before β (since β will be re-ordered before α). Typically, α would contain a read event r and β a write event w such that re-ordering the two enables r to read from w. Ensuring soundness and avoiding redundancy, i.e., exploring the same history multiple times, may require restricting the application of such re-orderings. This is modeled by the Boolean condition called OPTIMALITY. If this condition holds, the new explored histories are computed by the function SWAP. This function returns local states as well, which are necessary for continuing the exploration. We assume that $\text{SWAP}(h_<, \alpha, \beta, \mathsf{locals})$ returns pairs $(h'_{<'}, \mathsf{locals}')$ such that

1. h' contains at least the events in α and β,
2. h' without the events in α is a prefix of h, and
3. if a read r in α reads from different writes in h and h' (the wr relations of h and h' associate different transactions to r), then r is the last event in its transaction (w.r.t. po).

The first condition makes the re-ordering "meaningful" while the last two conditions ensure that the history h' is feasible by construction, i.e., it can be obtained using the operational semantics defined in Section 3.2.2. Feasibility of h' is ensured by keeping prefixes of transaction logs from h and all their wr dependencies except possibly for read events in α (second condition). In particular, for events in β, it implies that h' contains all their $(\mathsf{po} \cup \mathsf{so} \cup \mathsf{wr})^*$ predecessors. Also, the change of a read-from dependency is restricted to the last read in a transaction (third condition) because changing the value returned by a read may disable later events in the same transaction[1].

A concrete implementation of EXPLORE is called:

- *sound w.r.t.* ι if it outputs only histories in $\mathsf{hist}_\iota(\mathsf{P})$ for every program P,
- *complete w.r.t.* ι if it outputs every history in $\mathsf{hist}_\iota(\mathsf{P})$ for every program P,
- *optimal* if it does not output the same history twice,
- *strongly optimal* if it is optimal and never engages in fruitless explorations, i.e., EXPLORE is never called (recursively) on a history h that does not satisfy ι, and every call to EXPLORE results in an output or another recursive call to EXPLORE.

3.5 Swapping-Based Model Checking Algorithms for Prefix-Closed and Causally-Extensible Isolation Levels

We define a concrete implementation of EXPLORE, denoted as EXPLORE-CE, that is sound w.r.t. ι, complete w.r.t. ι, and strongly optimal for any isolation level ι that is prefix-closed and causally-extensible. The isolation level ι is a parameter of EXPLORE-CE. The space complexity of EXPLORE-CE is polynomial in the size of the program. An important invariant of this implementation is that it explores histories with *at most one* pending transaction and this transaction is maximal in session order. This invariant is used to avoid fruitless explorations: since ι is assumed to be causally-extensible, there always exists an extension of the current history with one more event that continues to satisfy ι. Moreover, this invariant is sufficient to guarantee completeness in the sense defined above of exploring all histories of "full" program executions (that end in a final configuration).

[1]Different wr dependencies for previous reads can be explored in other steps of the algorithm.

Section 3.5.1 describes the implementations of Next and ValidWrites used to extend a given execution, Section 3.5.2 describes the functions ComputeReorderings and Swap used to compute re-ordered executions, and Section 3.5.3 describes the Optimality restriction on re-ordering. We assume that the function Valid is defined as simply $\text{Valid}(h) ::= \mathsf{true}$ (no filter before outputting). Section 3.5.4 discusses correctness arguments.

3.5.1 Extending Histories According to An Oracle Order

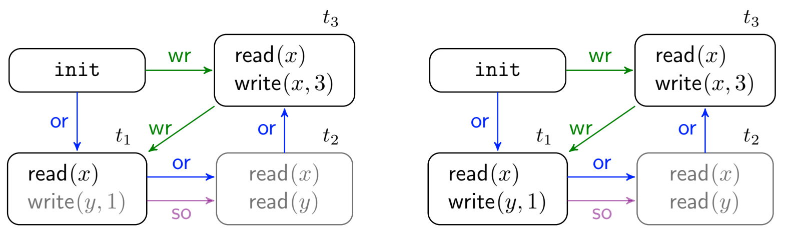
The function Next generates events representing database accesses to extend an execution, according to an *arbitrary but fixed* order between the transactions in the program called *oracle order*. We assume that the oracle order, denoted by $<_{\mathsf{or}}$, is consistent with the order between transactions in the same session of the program. The extension of $<_{\mathsf{or}}$ to events is defined as expected. For example, assuming that each session has an id, an oracle order can be defined by an order on session ids along with the session order so: transactions from sessions with smaller ids are considered first and the order between transactions in the same session follows so.

Next returns a new event of the transaction that is not already completed and that is *minimal* according to $<_{\mathsf{or}}$. In more detail, if j, e, γ is the output of $\text{Next}(\mathsf{P}, h_<, \mathsf{locals})$, then either:

- the last transaction log t of session j (w.r.t. so) in h is pending, and t is the smallest among pending transaction logs in h w.r.t. $<_{\mathsf{or}}$
- h contains no pending transaction logs and the next transaction of sessions j is the smallest among not yet started transactions in the program w.r.t. $<_{\mathsf{or}}$.

```
begin;              ||
a = read(x);        ||
if(a == 3)          ||  begin;
 write(y,1);        ||  d = read(x);
commit              ||  write(x,3);
begin;              ||  commit
b = read(x);        ||
c = read(y);        ||
commit              ||
```

(a) Program (2 sessions).



(b) An incomplete history. (c) An extension.

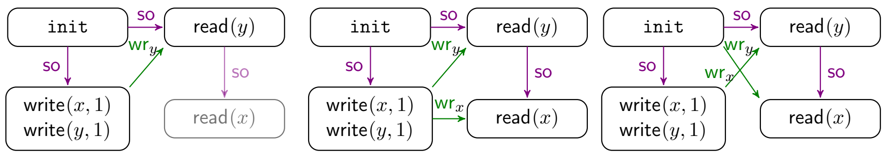
Figure 3.7: A program with two sessions (a), a history h (b), and an extension of h with an event returned by Next (c). The so-edges from `init` to the other transactions are omitted for legibility. We use edges labeled by or to represent the oracle order $<_{\mathsf{or}}$. Events in gray are not yet added to the history.

This implementation of Next is deterministic and it prioritizes the completion of pending transactions. The latter is useful to maintain the invariant that any history explored by the algorithm has at most one pending transaction. Preserving this invariant requires that the histories given as input to Next also have at most one pending transaction. This is discussed further when explaining the process of re-ordering events in Section 3.5.2.

For example, consider the program in Figure 3.7a, an oracle order which orders the two transactions in the left session before the transaction in the right session, and the history h in Figure 3.7b. Since the local state of the pending transaction on the left stores 3 to the local variable a (as a result of the previous $\texttt{read}(x)$ event) and the Boolean condition in `if` holds, Next returns the event $\mathsf{write}(y, 1)$ when called with h.

```
                 ‖ begin;
begin;           ‖ a = read(y);
write(x,1);      ‖ commit
write(y,1);      ‖ begin;
commit           ‖ b = read(x);
                 ‖ commit
```

(a) Program (2 sessions).



(b) Current history. (c) One extension. (d) Another extension.

Figure 3.8: Extensions of a history by adding a `read` event. Events in gray are not yet added to the history.

According to Algorithm 1, if the event returned by Next is not a read event, then it is simply added to the current history as the maximal element of the order $<$ (cf. the definition of $\oplus_j$ on ordered histories). If it is a read event, then adding this event may result in multiple histories depending on the chosen wr dependency. For example, in Figure 3.8, extending the history in Figure 3.8b with the $\texttt{read}(x)$ event could result in two different histories, pictured in Figure 3.8c and 3.8d, depending on the write with whom this read event is associated by wr. However, under `TCC`, the latter history is inconsistent. The function ValidWrites limits the choices to those that preserve consistency with the intended isolation level ι, i.e.,

$$\textsc{ValidWrites}(h, e) \coloneqq \{t \ \in \mathsf{cmtt}(h) \mid h \oplus_j e \oplus \mathsf{wr}(t, e) \text{ is consistent w.r.t. } \iota\}$$

where $\mathsf{cmtt}(h)$ is the set of committed transactions in h.

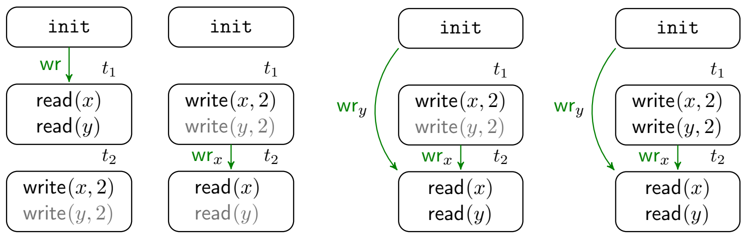
3.5.2 Re-Ordering Events in Histories

After extending the current history with one more event, explore may be called recursively on other histories obtained by re-ordering events in the current one (and dropping some other events).

Re-ordering events must preserve the invariant of producing histories with at most one pending transaction. To explain the use of this invariant in avoiding fruitless explorations, let us consider the program in Figure 3.9a assuming an exploration under Read Committed. The oracle order gives priority to the transaction on the left. Assume that the current history reached by the exploration is the one pictured in Figure 3.9b (the last added event is $\mathsf{write}(x, 2)$). Swapping $\mathsf{write}(x, 2)$ with $\mathsf{read}(x)$ would result in the history pictured in Figure 3.9c. To ensure that this swap produces a new history which was not explored in the past, the wr_x dependency of $\mathsf{read}(x)$ is changed towards the $\mathsf{write}(x, 2)$ transaction (we detail this later). By the definition of next (and the oracle order), this history shall be extended with $\mathsf{read}(y)$, and this read event will be associated by wr_y to the only available $\texttt{write}(y, \_)$ event from `init`. This is pictured in Figure 3.9d. The next exploration step will extend the

```
begin;          || begin;
a = read(x);    || write(x,2);
b = read(y);    || write(y,2);
commit          || commit
```

(a) Program (2 sessions).



(b) Current. (c) Reorder. (d) Extended. (e) Inconsistent.

Figure 3.9: Example of inconsistency after swapping two events. All so-edges from `init` to the other transactions are omitted for legibility. The history order $<$ is represented by the top to bottom order in each figure. Events in gray are not yet added to the history.

history with **write**$(y, 2)$ (the only extension possible) which however, results in a history that does *not* satisfy Read Committed, thereby, the recursive exploration branch being blocked. The core issue is related to the history in Figure 3.9d which has a pending transaction that is *not* $(\mathsf{so} \cup \mathsf{wr})^+$-maximal. Being able to extend such a transaction while maintaining consistency is not guaranteed by Read Committed (and any other isolation level we consider). Nevertheless, causal extensibility guarantees the existence of an extension for pending transactions that are $(\mathsf{so} \cup \mathsf{wr})^+$-maximal. We enforce this requirement by restricting the explored histories to have at most one pending transaction. This pending transaction will necessarily be $(\mathsf{so} \cup \mathsf{wr})^+$-maximal.

To enforce histories with at most one pending transaction, the function ComputeReorderings, which identifies events to reorder, has a non-empty return value only when the last added event is `commit` (the end of a transaction)[2]. Therefore, in such a case, it returns pairs of some transaction log prefix ending in a read r and the last completed transaction log t, such that the transaction log containing r and t are *not* causally dependent (i.e., related by $(\mathsf{so} \cup \mathsf{wr})^*$) (the transaction log prefix ending in r and t play the role of the subsequences α and respectively, β in the description of ComputeReorderings from Section 3.4). To simplify the notation, we will assume that ComputeReorderings returns pairs (r, t).

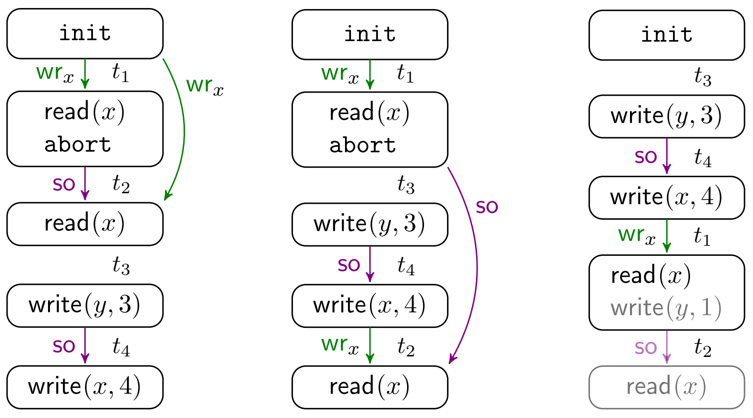
$$\textsc{ComputeReorderings}(h_<) := \left\{ (r,t) \in \mathsf{Events} \times T \;\middle|\; \begin{array}{l} t \text{ is complete } \land \\ t \text{ includes the last event in } < \land \\ r \in \mathsf{reads}(T) \land\ t \text{ writes } \mathsf{var}(r) \land \\ \mathsf{tr}(r) < t\ \land (\mathsf{tr}(r), t) \notin (\mathsf{so} \cup \mathsf{wr})^* \end{array} \right\}$$

For example, for the program in Figure 3.10a and the history h in Figure 3.10b, ComputeReorderings(h) would return (r_1, t_4) and (r_2, t_4) where r_1 and r_2 are the read(x) events in t_1 and t_2 respectively.

[2]Aborted transactions have no visible effect on the state of the database so swapping an aborted transaction cannot produce a new meaningful history.

```
begin;           ||
a = read(x);     ||
if(a == 0){      ||  begin;
     abort       ||  write(y,3);
}                ||  commit
write(y,1)       ||  begin;
commit           ||  write(x,4);
begin;           ||  commit
b = read(x);     ||
commit           ||
```

(a) Program (2 sessions).



(b) Current. (c) Swap t_2 and t_4. (d) Swap t_1 and t_4.

Figure 3.10: Re-ordering events. All so-edges from `init` to other transactions are omitted for legibility. The history order $<$ is represented by the top to bottom order in each figure. Events in gray are deleted from the history.

For a pair (r, t), the function Swap produces a new history h' which contains all the events ordered before r (w.r.t. $<$), the transaction t and all its $(\mathsf{so} \cup \mathsf{wr})^*$ predecessors, and the event r reading from t. All the other events are removed. Note that the po predecessors of r from the same transaction are ordered before r by $<$ and they will be also included in h'. The history h' without r is a prefix of the input history h. By definition, the only pending transaction in h' is the one containing the read r. The order relation is updated by moving the transaction containing the read r to be the last; it remains unchanged for the rest of the events.

$$\textsc{Swap}(h_<, r, t, \mathsf{locals}) := \big((h' = (h \setminus D) \oplus \mathsf{wr}(t, r), <'), \mathsf{locals}'\big)$$

where

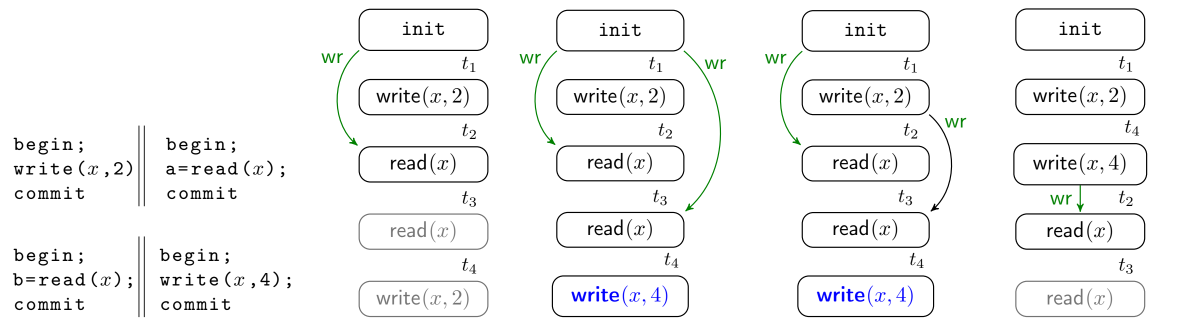
$$\begin{aligned} \mathsf{locals}' &= \mathsf{locals} \downarrow \mathsf{events}(h') \\ D &= \{e \mid r < e \ \wedge\ (\mathsf{tr}(e), t) \notin (\mathsf{so} \cup \mathsf{wr})^*\} \text{ and} \\ <' &= \big(<\downarrow (\mathsf{events}(h') \setminus \mathsf{events}(\mathsf{tr}(r)))\big) \cdot_D \mathsf{tr}(r) \end{aligned}$$

Above, $h \setminus D$ is the prefix of h obtained by deleting all the events in D from its transaction logs; a transaction log is removed altogether if it becomes empty. Also, $h'' \oplus \mathsf{wr}(t, r)$ denotes an *update* of the wr relation of h'' where any pair $(\_, r)$ is replaced by (t, r). Finally, the relation $<'' \cdot_D \mathsf{tr}(r)$ is an extension of the total order $<''$ obtained by appending the events in $\mathsf{events}(\mathsf{tr}(r))$ that are not in D according to program order.

Continuing with the example of Figure 3.10, when swapping r_1 and t_4, all the events in transaction t_2 belong to D and they will be removed. This is shown in Figure 3.10d. Note that transaction t_1 aborted in Figure 3.10b while it will commit in Figure 3.10d (because the value read from x changed). When swapping r_2 and t_4, no event but the commit in t_2 will be deleted (Figure 3.10c).

3.5.3 Ensuring Optimality

Simply extending histories according to NEXT and making recursive calls on re-ordered histories whenever they are consistent w.r.t. ι guarantees soundness and completeness, but it does not guarantee optimality. Intuitively, the source of redundancy is related to the fact that applying SWAP on different histories may give the same result.



```
begin;          || begin;
write(x,2)      || a=read(x);
commit          || commit

begin;          || begin;
b=read(x);      || write(x,4);
commit          || commit
```

(a) Program (4 sessions). (b) Current. (c) t_3 reads `init`. (d) t_3 reads t_1. (e) After swap.

Figure 3.11: Re-ordering events versus optimality. We assume an oracle order orders transaction from left to right, top to bottom in the program. All transaction logs are history-ordered top to bottom according to their position in the figure. Events in gray are not yet added to the history.

As a first example, consider the program in Figure 3.11a with 2 transactions that only read some variable x and 2 transactions that only write to x, each transaction in a different session. Assume that EXPLORE reaches the ordered history in Figure 3.11b and NEXT is about to return the second reading transaction. EXPLORE will be called recursively on the two histories in Figure 3.11c and Figure 3.11d that differ in the write that this last read is reading from (the initial write or the first write transaction). On both branches of the recursion, NEXT will extend the history with the last write transaction written in blue bold font. For both histories, swapping this last write with the first read on x will result in the history in Figure 3.11e (cf. the definition of COMPUTEREORDERINGS and SWAP). Thus, both branches of the recursion will continue extending the same history and optimality is violated. The source of non-optimality is related to wr dependencies that are *removed* during the SWAP computation. The histories in Figure 3.11c and Figure 3.11d differ in the wr dependency involving the last read, but this difference was discarded during the SWAP computation. To avoid this behavior, SWAP is enabled only on histories where the discarded wr dependencies relate to some "fixed" set of writes, i.e., latest[3] writes w.r.t. $<$ that guarantee consistency by causal extensibility (see the definition of $\mathsf{readLatest}_\iota(\_,\_,\_)$ below). By causal extensibility, a read r can always read from a write which already belongs to its "causal past", i.e., predecessors in $(\mathsf{so} \cup \mathsf{wr})^*$ excluding the wr dependency for r. For every discarded wr dependency, it is required that the read reads from the latest such write w.r.t. $<$. In this example, re-ordering

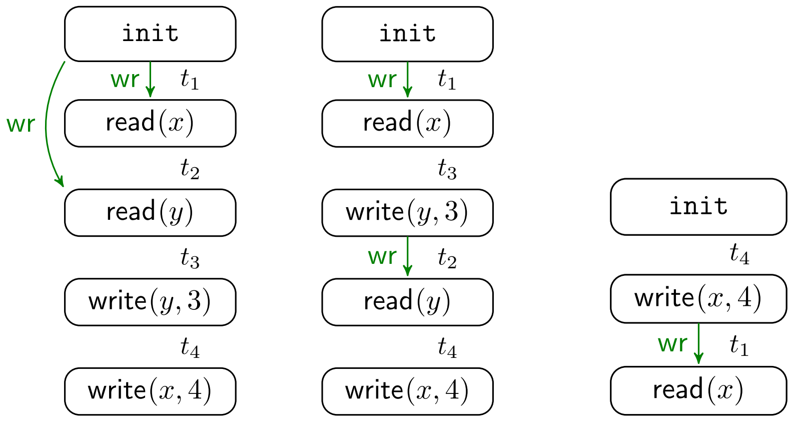
[3]We use latest writes because they are uniquely defined. In principle, other ways of identifying some unique set of writes could be used.

is enabled only when the second $\texttt{read}(x)$ reads from the initial write; $\texttt{write}(x, 2)$ does not belong to its "causal past" (when the wr dependency of the read itself is excluded).

```
begin;        || begin;
a=read(x);    || b=read(y);
commit        || commit

begin;        || begin;
write(y,3);   || write(x,4);
commit        || commit
```

(a) Program (4 sessions).



(b) Current history. (c) Swap t_2 and t_3. (d) Swap t_1 and t_4.

Figure 3.12: Re-ordering the same read on different branches of the recursion.

The restriction above is not sufficient, because the two histories for which SWAP gives the same result may not be generated during the same recursive call (for different wr choices when adding a read). For example, consider the program in Figure 3.12a that has four sessions each containing a single transaction. EXPLORE may compute the history h pictured in Figure 3.12b. Before adding transaction t_4, EXPLORE can re-order t_3 and t_2 and then extend with t_4 and arrive at the history h_1 in Figure 3.12c. Also, after adding t_4, it can re-order t_1 and t_4 and arrive at the history h_2 in Figure 3.12d. However, swapping the same t_1 and t_4 in h_1 leads to the same history h_2, thereby, having two recursive branches that end up with the same input and violate optimality. Swapping t_1 and t_4 in h_1 should not be enabled because the $\texttt{read}(y)$ to be removed by SWAP has been swapped in the past. Removing it makes it possible that this recursive branch explores that wr choice for $\texttt{read}(y)$ again.

The OPTIMALITY condition restricting re-orderings requires that the re-ordered history be consistent w.r.t. ι and that every read deleted by SWAP or the re-ordered read r (whose wr dependency is modified) reads from a latest valid write, cf. the example in Figure 3.11, and it is not already swapped, cf. the example in Figure 3.12 (the set D is defined as in SWAP):

$$\textsc{Optimality}(h_<, r, t, \mathsf{locals}) := \text{the history returned by } \textsc{Swap}(h_<, r, t, \mathsf{locals}) \text{ satisfies } \iota \wedge \\ \forall r' \in \mathsf{reads}(h) \cap (D \cup \{r\}).\ \neg\textsc{swapped}(h_<, r') \wedge \\ \mathsf{readLatest}_\iota(h_<, r', t)$$

A read r reads from a causally latest valid transaction, denoted as $\mathsf{readLatest}_\iota(h_<, r,)$, if reading from any other later transaction t' w.r.t. $<$ which is in the "causal past" of $\mathsf{tr}(r)$ in $h_<$ violates the isolation level ι. Formally, assuming that t_r is the transaction such that $(t_r, r) \in \mathsf{wr}$ in h,

$$\mathsf{readLatest}_\iota(h_<, r, t) := t_r = \max_< \left\{ \begin{array}{c} t' \text{ writes } \mathsf{var}(r) \wedge (t', \mathsf{tr}(r)) \in (\mathsf{so} \cup \mathsf{wr})^* \text{ holds in } h' \\ \wedge\ h' \oplus r \oplus \mathsf{wr}(t', r) \text{ is consistent w.r.t. } \iota \end{array} \right\}$$

where $h' = h \setminus \{e \mid r \leq e \wedge (\mathsf{tr}(e), t) \notin (\mathsf{so} \cup \mathsf{wr})^*\}$.

We say that a read r is *swapped* in $h_<$ when (1) r reads from a transaction t that is a successor in the oracle order $<_{\mathsf{or}}$ (the transaction was added by NEXT after the read), which is now a predecessor[4] in the history order $<$, (2) there is no transaction t' that is before r in both $<_{\mathsf{or}}$ and $<$, and which is a $(\mathsf{so} \cup \mathsf{wr})^+$ successor of t, and (3) r is the first read in its transaction to read from t. Formally:

$$\begin{aligned}\textsc{swapped}(h_<, r) \coloneqq t < r \wedge t >_{\mathsf{or}} r \;\wedge\; \forall t' \in h.\ t' <_{\mathsf{or}} \mathsf{tr}(r) \implies (r < t' \vee (t, t') \notin (\mathsf{so} \cup \mathsf{wr})^+) \\ \wedge\; \forall r' \in \mathsf{reads}(h).\ (t, r') \in \mathsf{wr} \implies (r', r) \notin \mathsf{po}\end{aligned}$$

where $t = \mathsf{wr}^{-1}(r)$.

Condition (1) states a quite straightforward fact about swaps: r could not have been involved in a swap if it reads from a predecessor in the oracle order which means that it was added by NEXT after the transaction it reads from. Conditions (2) and (3) are used to exclude spurious classifications as swapped reads. Concerning condition (2), suppose that in a history h we swap a transaction t with respect a (previous) read event r. Later on, the algorithm may add a read r' reading also from t. Condition (2) forbids r' to be declared as swapped. Indeed, taking $\mathsf{tr}(r)$ as an instantiation of t', $\mathsf{tr}(r)$ is before r' in both $<_{or}$ and $<$ and it reads from the same transaction as r', thereby, being a $(\mathsf{so} \cup \mathsf{wr})^+$ successor of the transaction read by r'. Condition (3) forbids that, after swapping r and t in h, later read events from the same transaction as r can be considered as swapped.

Showing that completeness w.r.t. ι holds despite discarding re-orderings is quite challenging. Intuitively, it can be shown that if some SWAP is *not* enabled in some history $h_<$ for some pair (r, t) although the result would be consistent w.r.t. ι (i.e., OPTIMALITY$(h_<, r, t, \mathsf{locals})$ does not hold because some deleted read is swapped or does not read from a causally latest transaction), then the algorithm explores another history h' which coincides with h except for those deleted reads who are now reading from causally latest transactions. Then, h' would satisfy OPTIMALITY$(h_<, r, t, \mathsf{locals})$, and moreover applying SWAP on h' for the pair (r, t) would lead to the same result as applying SWAP on h, thereby, ensuring completeness.

3.5.4 Correctness

The following theorem states the correctness of the algorithm presented in this section:

Theorem 3.5.1. *For any prefix-closed and causally-extensible isolation level ι,* EXPLORE-CE *is sound w.r.t. ι, complete w.r.t. ι, strongly optimal, and employs polynomial space.*

The soundness of EXPLORE-CE w.r.t. ι is a consequence of the VALIDWRITES and OPTIMALITY definitions which guarantee that all histories given to recursive calls are consistent w.r.t. ι, and of the SWAP definition which ensures to only produce feasible histories (which can be obtained using the operational semantics defined in Section 3.2.2). The fact that this algorithm never engages in fruitless explorations follows easily from causal-extensibility which

[4]The EXPLORE maintains the invariant that every read follows the transaction it reads from in the history order $<$.

ensures that any current history can be extended with any event returned by NEXT. Polynomial space is also quite straightforward since the **for all** loops in Algorithm 1 have a linear number of iterations: the number of iterations of the loop in EXPLORE, resp., EXPLORESWAPS, is bounded by the number of write, resp., read, events in the current history (which is smaller than the size of the program; recall that we assume bounded programs with no loops as usual in SMC algorithms). On the other hand, the proofs of completeness w.r.t. ι and optimality are quite complex.

The completeness of EXPLORE-CE w.r.t. ι means that for any given program P, the algorithm outputs every history h in $\mathsf{hist}_\iota(\mathsf{P})$. The proof of completeness w.r.t. ι (Theorem 3.5.19) defines a sequence of histories produced by the algorithm starting with an empty history and ending in $\overline{h}$, for every consistent w.r.t. ι history $\overline{h}$. It consists of several steps:

1. Defining a *canonical* total order $<^h$ for every unordered partial history h, such that if the algorithm reaches the ordered history $h_<$, then $<$ and $<^h$ coincide (Lemmas 3.5.6 and 3.5.11). This canonical order is useful in future proof steps as it allows to extend several definitions to arbitrary histories that are not necessarily reachable, such as OPTIMALITY or SWAPPED predicates, by evaluating them on the ordered history using the canonical order.

2. Defining the notion of *oracle-respectfulness*, an invariant satisfied by every (partial) ordered history reached by the algorithm. Briefly, a history is oracle-respectful if it has only one pending transaction and for every two events e, e' such that $e <_{\mathsf{or}} e'$, either $e < e'$ or there is a swapped event e'' in between. We prove in Lemma 3.5.9 that every reachable history is oracle-respectful. Oracle-respectfulness allow us proving that for every history h and e, SWAPPED(h, e) holds iff e has been swapped in the computable path reaching h.

3. Defining a deterministic function PREV, the *previous* function, which takes as input a partial history (not necessarily reachable) and returns a partial history. We show in Lemma 3.5.16 that if h is reachable, then PREV(h) returns the history computed by the algorithm just before h (i.e., the previous history in the call stack). We also prove that if a history h oracle-respectful, then PREV(h) is also oracle-respectful (Lemma 3.5.13).

4. Deducing that in particular, as $\overline{h}$ is reachable, it is oracle-respectful. Then, there is a finite collection of oracle-respectful histories $H_h = \{h_i\}_{i=0}^{n}$ such that $h_n = h$, $h_0 = (\mathtt{init}, \emptyset, \emptyset)$, $h_n = \overline{h}$ and $h_i = $ PREV(h_{i+1}) (Lemma 3.5.18). The oracle-respectfulness invariant is key to being able to construct such a collection. In particular, they are used to prove that h_i has at most the same number of swapped events as h_{i+1} and in case of equality, h_i contain exactly one event less than h_{i+1} (Lemma 3.5.16), which implies that the collection is indeed finite (Lemma 3.5.17).

5. Proving that for every history h that is oracle-respectful and consistent w.r.t. ι and PREV(h) is reachable, then h is also reachable (Lemma 3.5.14). Conclude by induction that every history in H_h is reachable, as h_0 is the initial state and $h_i = $ PREV(h_{i+1}) Lemma 3.5.18.

The proof of strong optimality relies on arguments employed for proving completeness w.r.t. ι. It can be shown that if the algorithm would reach a (partial) history h twice, then for one of the two exploration branches, the history h' computed just before h would be different from $\textsc{prev}(h)$, which contradicts the definition of $\textsc{prev}(h)$.

In terms of time complexity, the $\textsc{explore-ce}(\iota)$ algorithm achieves polynomial time between consecutive outputs for isolation levels ι where checking consistency w.r.t. ι of a history is polynomial time, e.g., RC, RA, and TCC.

3.5.4.1 Canonical Order of a History

We define a total order for every history that coincides on reachable histories with the history order. For achieving it, we analyze how the algorithm orders two transaction t, t' in an ordered history $h_<$. In what follows, we consider an *unordered* history $h = (T, \mathsf{so}, \mathsf{wr})$; where T, so and wr represent the set of transactions, session-order and write-read dependencies of h. On one hand, if $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^*$, then $t < t'$. On the other hand, if t and t' are $(\mathsf{so} \cup \mathsf{wr})^*$-incomparable, the algorithm prioritizes the one that is read by a smaller `read` event according to or. Combining both arguments recursively we obtain a *canonical order* for a history, which is formally defined with the function presented below.

Algorithm 3 Canonical order

```
1:  function canonicalOrder($(h, t, t')$)
2:    return $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^* \lor$
3:           $((t', t) \notin (\mathsf{so} \cup \mathsf{wr})^* \land \textsc{minimalDependency}(h, t, t', \bot))$
4:  function minimalDependency($(h, t, t', e)$)
5:    let $a = \min_{<_{\mathsf{or}}} \textsc{dep}(h, t, e)$; $a' = \min_{<_{\mathsf{or}}} \textsc{dep}(h, t', e)$
6:    if $a \neq a'$ then
7:      return $a <_{\mathsf{or}} a'$
8:    else
9:      return $\textsc{minimalDependency}(h, t, t', a)$
10: function dep($(h, t, e)$)
11:   return $R(h, t, e) \setminus \{e\} \cup \{e' \mid e' \in t\}$; where
12:     $R(h, t, e) = \{r \mid \exists t' \in T.(t, t') \in (\mathsf{so} \cup \mathsf{wr})^* \land (t', r) \in \mathsf{wr} \land (\mathsf{tr}(r), \mathsf{tr}(e)) \in (\mathsf{so} \cup \mathsf{wr})^*\}$
```

The function $\textsc{canonicalOrder}$ produces a relation between transactions in a history, denoted $\leq^h$. In algorithm 3's description, we denote $\bot$ to represent the end of the program, which always exists, and that is so-related with every single transaction.

Firstly, we prove in Corollary 3.5.4 that the canonical order is well-defined for every pair of transactions in h (i.e. for every pair of distinct transactions $t, t' \in T$, $\textsc{canonicalOrder}(h, t, t')$ halts). The proof requires proving that $\textsc{minimalDependency}(h, t, t', \bot)$ halts (Lemma 3.5.3). For achieving it, we show in Lemma 3.5.2 that the intermediate values a and a' computed by $\textsc{dep}$ during recursive calls to function $\textsc{minimalDependency}$ are linked.

Lemma 3.5.2. *Let e be an event and let $t \in T$ be a transaction. $\textsc{dep}(h, t, r') \subseteq \textsc{dep}(h, t, e)$; where $r' = \min_{<_{\mathsf{or}}} \textsc{dep}(h, t, e)$. Moreover, if $r' \notin t$, the inclusion is strict.*

Proof. For proving the result, we distinguish whether $\textsc{dep}(h, t, e)$ is $\{e' \mid e' \in t\}$ or not. On one hand, if $\textsc{dep}(h, t, e) = \{e' \mid e' \in t\}$, the lemma is immediately holds. On the other hand, if $\textsc{dep}(h, t, e) \neq \{e' \mid e' \in t\}$, let us consider $r \in \textsc{dep}(h, t, r')$. We can assume w.l.o.g. that $r \notin t$; as otherwise, r trivially belongs to $\textsc{dep}(h, t, e)$. In such case, there exists a transaction $t' \in T$ s.t. $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t', r) \in \mathsf{wr}$ and $(\mathsf{tr}(r), \mathsf{tr}(r')) \in (\mathsf{so} \cup \mathsf{wr})^*$. Moreover, as $r' \in \textsc{dep}(h, t, e)$, there exists a transaction $t'' \in T$ s.t. $(t, t'') \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t'', r') \in \mathsf{wr}$ and $(\mathsf{tr}(r'), \mathsf{tr}(e)) \in (\mathsf{so} \cup \mathsf{wr})^*$. In particular, as $(\mathsf{tr}(r), \mathsf{tr}(r')) \in (\mathsf{so} \cup \mathsf{wr})^*$ and $(\mathsf{tr}(r'), \mathsf{tr}(e)) \in (\mathsf{so} \cup \mathsf{wr})^*$, we deduce that $(\mathsf{tr}(r), \mathsf{tr}(e)) \in (\mathsf{so} \cup \mathsf{wr})^*$. Altogether, we conclude that $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t', r) \in \mathsf{wr}$ and $(\mathsf{tr}(r'), \mathsf{tr}(e)) \in (\mathsf{so} \cup \mathsf{wr})^*$; i.e. $r \in \textsc{dep}(h, t, e)$. The moreover comes trivially as if $r' \notin t$, $r' \notin \textsc{dep}(h, t, r')$. □

Lemma 3.5.3. *For every pair of distinct transactions $t, t' \in T$, the function* $\textsc{minimalDependency}(h, t, t', \bot)$ *always halts.*

Proof. We reason by contrapositive, assuming that $\textsc{minimalDependency}(h, t, t', \bot)$ does not halt. In such case, there exists an infinite chain of events $e_n, n \in \mathbb{N}$ such that $e_0 = \bot$, and for every $n \in \mathbb{N}$, $e_{n+1} = \min_{\mathsf{or}} \textsc{dep}(h, t, e_n) = \min_{\mathsf{or}} \textsc{dep}(h, t', e_n)$. First, as h is finite, so are both $\textsc{dep}(h, t, e_n)$ and $\textsc{dep}(h, t', e_n)$. Also, if $e_n \notin t$, by Lemma 3.5.2, $\textsc{dep}(h, t, e_{n+1}) \subsetneq \textsc{dep}(h, t, e_n)$ (and analogously for t'). Hence, the number of events e_n that are neither in t nor in t' is finite. Let k be the minimum integer s.t. $e_k \in t \cup t'$. Let us assume w.l.o.g. that $e_k \in t$. In such case, by Lemma 3.5.2, for every $n \in \mathbb{N}$ s.t. $n \geq k$, $\textsc{dep}(h, t, e_n) \subseteq \textsc{dep}(h, t, e_k)$; so $e_k \leq_{\mathsf{or}} e_n$. However, as $e_k \in t$, $e_k \in \textsc{dep}(h, t, e_n)$; so $e_k = e_n$. In particular, thanks to Lemma 3.5.2, we deduce that $e_k \in t'$ as well. As no event belong to two transactions at the same time, $t = t'$. However, this is impossible as $t \neq t'$; so the initial hypothesis, that $\textsc{minimalDependency}(h, t, t', \bot)$ does not halt, is false. □

The following result is just a consequence of Lemma 3.5.3 and the definition of $\textsc{canonicalOrder}$.

Corollary 3.5.4. *The relation $\leq^h$ is well-defined for every pair of distinct transactions.*

Next, we prove in Lemma 3.5.6 that $\leq^h$ is a total order. In particular, for proving that $\leq^h$ is transitive, we prove an auxiliary result that characterizes $(\mathsf{so} \cup \mathsf{wr})^*$-dependent transactions.

Lemma 3.5.5. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history and let $t, t' \in T$ be a pair of transactions that $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^*$. One of the following conditions holds:*

1. *$t \leq_{\mathsf{or}} t'$, or*
2. *$\exists t'', t''' \in T$ s.t. $(t, t'') \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t'', t''') \in \mathsf{wr}$, $(t''', t') \in (\mathsf{so} \cup \mathsf{wr})^*$ and $t''' \leq_{\mathsf{or}} t'$.*

Proof. For proving the result, let us assume that $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^*$ but that $t' <_{\mathsf{or}} t$ and let us prove that condition 2 holds. For that, let us consider the sets of pairs of transactions $A(t, t')$ and $B(t, t')$ defined as follows:

$$A(t, t') \coloneqq \{(t'', t'''') \in T^2 \mid (t, t'') \in (\mathsf{so} \cup \mathsf{wr})^* \land (t'', t''') \in \mathsf{wr} \land (t''', t') \in (\mathsf{so} \cup \mathsf{wr})^*\} \quad (3.2)$$

$$B(t, t') \coloneqq \{(t'', t'''') \in A(t, t') \mid t''' \leq_{\mathsf{or}} t'\} \quad (3.3)$$

As $t' <_{\mathsf{or}} t$ and or respects so, $t \neq t'$ and $(t, t') \notin \mathsf{so}$. Hence, $A(t, t') \neq \emptyset$. To conclude 2, it suffices to prove that $B(t, t') \neq \emptyset$. By contrapositive, let us assume that $B(t, t') = \emptyset$. As $A(t, t') \neq \emptyset$, let $(t'', t''') \in A(t, t')$ be a pair of transactions s.t. t''' is $(\mathsf{so} \cup \mathsf{wr})^*$-maximal. As t' has finitely many $(\mathsf{so} \cup \mathsf{wr})^*$-predecessors, and for any pair of transactions $(t_1, t_2) \in A(t, t')$, t_2 is a $(\mathsf{so} \cup \mathsf{wr})^*$-predecessors of t'; such transaction t''' is well-defined. Then, as $B(t, t') = \emptyset$, $t' <_{\mathsf{or}} t'''$. However, as or is consistent with so and $(t''', t') \in (\mathsf{so} \cup \mathsf{wr})^*$, $t''' \neq t'$ and $(t''', t') \notin \mathsf{so}$. Hence, there exists transactions s'', s''' s.t. $(t''', s'') \in (\mathsf{so} \cup \mathsf{wr})^*$, $(s'', s''') \in \mathsf{wr}$ and $(s''', t') \in (\mathsf{so} \cup \mathsf{wr})^*$. We observe that as $(t'', t''') \in A(t, t')$, we deduce that $(t, s'') \in (\mathsf{so} \cup \mathsf{wr})^*$. Therefore, we conclude that $(s'', s''') \in A(t, t')$. However, this is impossible as t''' is $(\mathsf{so} \cup \mathsf{wr})^*$-maximal and $(t''', s''') \in (\mathsf{so} \cup \mathsf{wr})^+$. Altogether, we deduce that our assumption that $B(t, t') = \emptyset$ is false; so $B(t, t') \neq \emptyset$; which concludes the result. □

Lemma 3.5.6. *The relation $\leq^h$ is a total order.*

Proof.

- Strongly connection: Let $t_1, t_2 \in T$ s.t. $t_1 \not\leq^h t_2$. If $(t_2, t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$, then $t_2 \leq^h t_1$. Otherwise, as $t_1 \not\leq^h t_2$, then $(t_1, t_2) \notin (\mathsf{so} \cup \mathsf{wr})^*$ and $\textsc{minimalDependency}(h, t_1, t_2, \bot)$ does not hold. In such case, as $\textsc{minimalDependency}$ halts (Lemma 3.5.3) and $\textsc{minimalDependency}(h, t_1, t_2, \bot)$ does not hold, by the definition of $\textsc{minimalDependency}$, $\textsc{minimalDependency}(h, t_2, t_1, \bot)$ holds. Altogether, we deduce that $(t_1, t_2) \notin (\mathsf{so} \cup \mathsf{wr})^*$ and $\textsc{minimalDependency}(h, t_2, t_1, \bot)$ holds; so $t_2 \leq^h t_1$.

- Reflexivity: As $(t, t) \in (\mathsf{so} \cup \mathsf{wr})^*$, $t \leq^h t$.

- Transitivity: Let t_1, t_2, t_3 three distinct transactions such that $t_1 \leq^h t_2$ and $t_2 \leq^h t_3$. Clearly, if $(t_1, t_3) \in (\mathsf{so} \cup \mathsf{wr})^*$, then $t_1 \leq^h t_3$. Otherwise, we prove that $(t_3, t_1) \notin (\mathsf{so} \cup \mathsf{wr})^*$ and $\textsc{minimalDependency}(h, t_1, t_3, \bot)$ holds.

  For proving it, we construct an inductive sequence of events e_n^i, $n \in \mathbb{N}$, $i \in \{1, 2, 3\}$ that represent the recursive calls to $\textsc{minimalDependency}$ function. We define e_0^i as $\bot$, and $e_{n+1}^i = \min_{<_{\mathsf{or}}} \textsc{dep}(h, t_i, e_n^i)$. Four cases arise depending on the relation between t_1, t_2 and t_3:

  - $(t_1, t_2) \in (\mathsf{so} \cup \mathsf{wr})^*$ and $(t_2, t_3) \in (\mathsf{so} \cup \mathsf{wr})^*$: In this case, $(t_1, t_3) \in (\mathsf{so} \cup \mathsf{wr})^*$; which is impossible as $(t_1, t_3) \notin (\mathsf{so} \cup \mathsf{wr})^*$. Thus, this scenario is impossible.
  - $(t_1, t_2) \notin (\mathsf{so} \cup \mathsf{wr})^*$ and $(t_2, t_3) \in (\mathsf{so} \cup \mathsf{wr})^*$: First we prove that $(t_3, t_1) \notin (\mathsf{so} \cup \mathsf{wr})^*$. If (t_3, t_1) would be in $(\mathsf{so} \cup \mathsf{wr})^*$, $(t_2, t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$; so $t_2 \leq^h t_1$. However, this is impossible as $t_1 \leq^h t_2$ and $\leq^h$ is strongly connected.

    Next, we prove that $\textsc{minimalDependency}(h, t_1, t_3, \bot)$ holds. We observe that as $(t_2, t_3) \in (\mathsf{so} \cup \mathsf{wr})^*$, one of the conditions 1 and 2 in Lemma 3.5.5 hold. Thus, for every $n \in \mathbb{N}$, then $e_n^2 \leq_{\mathsf{or}} e_n^3$. Moreover, as $(t_1, t_2) \notin (\mathsf{so} \cup \mathsf{wr})^*$ but $t_1 \leq^h t_2$ holds, $\textsc{minimalDependency}(h, t_1, t_2, \bot)$ holds. Let thus n_0 be the maximum $n \in \mathbb{N}$ s.t. $e_n^1 = e_n^2$. By Lemma 3.5.3, we know that n_0 is well-defined. We observe that in such case, $e_{n_0+1}^1 <_{\mathsf{or}} e_{n_0+1}^2 \leq_{\mathsf{or}} e_{n_0+1}^3$. Hence, as for every $n \leq n_0$, $e_n^1 \leq_{\mathsf{or}} e_n^3$ and $e_{n_0+1}^1 <_{\mathsf{or}} e_{n_0+1}^3$, we conclude that $\textsc{minimalDependency}(h, t_1, t_3, \bot)$ holds.

– $(t_1, t_2) \in (\mathsf{so} \cup \mathsf{wr})^*$ and $(t_2, t_3) \notin (\mathsf{so} \cup \mathsf{wr})^*$: First we prove that $(t_3, t_1) \notin (\mathsf{so} \cup \mathsf{wr})^*$. If (t_3, t_1) would be in $(\mathsf{so} \cup \mathsf{wr})^*$, $(t_3, t_2) \in (\mathsf{so} \cup \mathsf{wr})^*$; so $t_3 \leq^h t_2$. However, this is impossible as $t_2 \leq^h t_3$ and $\leq^h$ is strongly connected.

  Next, we prove that $\textsc{minimalDependency}(h, t_1, t_3, \bot)$ holds. We observe that as $(t_1, t_2) \in (\mathsf{so} \cup \mathsf{wr})^*$, one of the conditions 1 and 2 in Lemma 3.5.5 hold. Thus, for every $n \in \mathbb{N}$, then $e^1_n \leq_{\mathsf{or}} e^2_n$. Moreover, as $(t_2, t_3) \notin (\mathsf{so} \cup \mathsf{wr})^*$ but $t_2 \leq^h t_3$ holds, $\textsc{minimalDependency}(h, t_2, t_3, \bot)$ holds. Let thus n_0 be the maximum $n \in \mathbb{N}$ s.t. $e^2_n = e^3_n$. By Lemma 3.5.3, we know that n_0 is well-defined. We observe that in such case, $e^1_{n_0+1} \leq_{\mathsf{or}} e^2_{n_0+1} <_{\mathsf{or}} e^3_{n_0+1}$. Hence, as for every $n \leq n_0$, $e^1_n \leq_{\mathsf{or}} e^3_n$ and $e^1_{n_0+1} <_{\mathsf{or}} e^3_{n_0+1}$, we conclude that $\textsc{minimalDependency}(h, t_1, t_3, \bot)$ holds.

– $(t_1, t_2) \notin (\mathsf{so} \cup \mathsf{wr})^*$ and $(t_2, t_3) \notin (\mathsf{so} \cup \mathsf{wr})^*$: For proving that $\textsc{minimalDependency}(h, t_1, t_3, \bot)$ holds, let n_0 be the maximum $n \in \mathbb{N}$ s.t. $e^1_n = e^2_n = e^3_n$. By Lemma 3.5.3, we know that n_0 is well-defined. In such case, as $t_1 \leq^h t_2$ and $t_2 \leq^h t_3$, we know that $e^1_{n_0+1} \leq_{\mathsf{or}} e^2_{n_0+1} \leq_{\mathsf{or}} e^3_{n_0+1}$. Altogether, as not all the three events are not equal, we deduce that $e^1_{n_0+1} <_{\mathsf{or}} e^3_{n_0+1}$. Hence, $\textsc{minimalDependency}(h, t_1, t_3, \bot)$ holds.

  For proving that $(t_3, t_1) \notin (\mathsf{so} \cup \mathsf{wr})^*$, we reason by contrapositive, assuming that $(t_3, t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$ and reaching a contradiction. In such case, as $e^1_{n_0+1} <_{\mathsf{or}} e^3_{n_0+1}$, by the definition of $\textsc{dep}$, $e^1_{n_0+1} \in t_1$. Hence, $\min_{\mathsf{or}} t_1 = e^1_{n_0+1} <_{\mathsf{or}} e^3_{n_0+1} \leq_{\mathsf{or}} \min_{\mathsf{or}} t_3$. In particular, as or is consistent with so, we deduce that $(t_3, t_1) \notin \mathsf{so}$. Therefore, as we assume that $(t_3, t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$, there exists a transaction t and an event r s.t. $(t_3, t) \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t, r) \in \mathsf{wr}$ and $(\mathsf{tr}(r), t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$. Hence, $e^1_{n_0+1} \in \textsc{dep}(h, t_3, e^3_{n_0})$; so $e^3_{n_0} \leq e^1_{n_0+1}$. However, this is impossible as $e^1_{n_0+1} <_{\mathsf{or}} e^3_{n_0+1}$; so $(t_3, t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$.

- Antisymmetric Let t_1, t_2 s.t. $t_1 \leq^h t_2$ and $t_2 \leq^h t_1$. If $(t_1, t_2) \notin (\mathsf{so} \cup \mathsf{wr})^*$, we would deduce that $(t_2, t_1) \notin (\mathsf{so} \cup \mathsf{wr})^*$ and $\textsc{minimalDependency}(h, t_1, t_2, \bot)$ returns true. Then, as $t_2 \leq^h t_1$ and $(t_2, t_1) \notin (\mathsf{so} \cup \mathsf{wr})^*$, we deduce that $\textsc{minimalDependency}(h, t_2, t_1, \bot)$ returns true. However, this is impossible by the definition of $\textsc{minimalDependency}$; so $(t_1, t_2) \in (\mathsf{so} \cup \mathsf{wr})^*$, In such case, as $t_2 \leq^h t_1$, $(t_2, t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$; so $t_1 = t_2$.

□

3.5.4.2 Oracle-Respectful Histories

We prove in this section that histories can be always assumed ordered without loss of generality. First, we show in Lemma 3.5.11 that on reachable histories $(h, <)$, the order $<$ coincides with its canonical order. Thus, as by Lemma 3.5.6, the canonical order is defined for any history and it coincides with the order obtained using Algorithm 1, histories can always be assumed as ordered. For proving it, we rely on the notion of *oracle-respectfulness* (Definition 3.5.7), that captures a sufficient condition for an order $<$ to mismatch the oracle order or and correspond to the order of a reachable history. This notion allows us proving in Lemma 3.5.12 that the canonical order on any total history is oracle-respectful; implying that total and reachable histories behave “similarly”. The latter property hints that total histories are reachable, i.e. that Algorithm 1 is complete.

Definition 3.5.7. *An ordered history $(h, \leq)$ is* oracle-respectful *if*

1. *h has at most one pending transaction log, the last one w.r.t. $\leq$, and*
2. *for every pair of events $e \in \mathsf{P}, e' \in h$ s.t. $e \leq_{\mathsf{or}} e'$,*
   (a) *$e \leq e'$, or*
   (b) *$\exists e'', t'' \in h$ s.t. $e'' \leq_{\mathsf{or}} e$, $(\mathsf{tr}(e'), t'') \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t'', e'') \in \mathsf{wr}$, $e'' \leq e$ and $\textsc{swapped}(h_<, e'')$ holds;*

   where if $e \notin h$ we state $e' \leq e$ always holds, but $e \leq e'$ never does.

We denote it by $\mathtt{R}^{\mathsf{or}}(h, \leq)$.

For reasoning about reachable histories, we use the notion of computable path. A *computable path* $p = \{((h_n, <_n), \mathsf{locals}_n)\}_n$ is a sequence of pairs of histories and local variables s.t. (1) every history is reachable, (2) $h_0 = (\mathtt{init}, \emptyset, \emptyset)$ and (3) h_n (respectively locals_n) corresponds to the ordered history (resp. the local variables) appearing in n-th recursive call to $\textsc{explore}$. We denote by $\mathsf{len}(p)$ to the number of histories p has.

Lemma 3.5.8. *Let $h_< = ((T, \mathsf{so}, \mathsf{wr}), <)$ be a reachable history. For every pair of transactions t, t', (1) t, t' are totally ordered by $<$ and (2) if $(t, t') \in (\mathsf{so} \cup \mathsf{wr})^*$, then $t \leq t'$.*

Proof. As $h_<$ is reachable, there exists a computable path p whose last transaction is $h_<$. We prove the result by induction on the histories in such path. The base case, h_0, trivially holds as h_0 only contains one transaction. Let us thus assume that the result holds for the history h_n, and let us prove it for history h_{n+1}. Let $j, e, \gamma = \textsc{Next}(\mathsf{P}, h_n, \mathsf{locals}_n)$. On one hand, if $h_{n+1} = h_n \oplus_j e$ or $h_{n+1} = h_n \oplus e \oplus \mathsf{wr}(t, e)$, clearly conditions (1) and (2) hold; where $t \in \textsc{ValidWrites}(h_n, e)$. On the other hand, if h_{n+1} is the history returned by $\textsc{Swap}(h_n \oplus je, r, t, \mathsf{locals}'_n)$; for some $(r, t) \in \textsc{ComputeReorderings}(h_n)$ and $\mathsf{locals}'_n = \mathsf{locals}_n[e \mapsto \gamma]$, the result also holds by $\textsc{ComputeReorderings}$ and $\textsc{Swap}$'s definitions. ☐

In general, arbitrary histories are not necessarily oracle-respectful. One instance of oracle-respectful histories are reachable histories.

Lemma 3.5.9. *Every reachable history $(h, \leq_h)$ is oracle-respectful.*

Proof. As $h_<$ is reachable, there exists a computable path p whose last transaction is $h_<$. We prove the result by induction on the histories in such path. The base case, when the history is h_0, is immediate as h_0 only contains one transaction, $\mathtt{init}$. Let us thus assume that the result holds for the pair $(h_n, \mathtt{local}_n)$, and let us prove it for the pair $(h_{n+1}, \mathsf{locals}_{n+1})$. We denote by $\leq_n$ and $\leq_{n+1}$ to the order of h_n and h_{n+1} resp. By hypothesis, h_n is oracle-respectful. We distinguish several cases, depending on the event a s.t. $(j, a, \gamma) = \textsc{next}(\mathsf{P}, h_n, \mathsf{locals}_n)$.

- a is a $\mathtt{begin}$ event: In this case, the last event in $<_{n+1}$ is a, and $\mathsf{tr}(a)$ is not complete. Hence, $\textsc{ComputeReorderings}(h_{n+1}) = \emptyset$; so $h_{n+1} = h_n \oplus_j a$. By the choice of $\textsc{next}$ function (see Section 3.5.1), we deduce that h_n has no pending transactions, so 1 holds. For showing 2, let $e \in \mathsf{P}, e' \in h_{n+1}$ s.t. $e <_{\mathsf{or}} e'$. As $a = \min_{\mathsf{or}} \mathsf{P} \setminus h_n$ there is no event $e \in \mathsf{P} \setminus h_n$ s.t. $e \leq_{\mathsf{or}}$. Hence, $e' \neq a$ or $e \in h_n$. In such case, as $\leq_{n+1}$ is an extension of $\leq_n$ and Property 2 holds for h_n; it also holds for h_{n+1}.

- $\underline{a \text{ is a } \texttt{read} \text{ event}}$: Similarly to the previous case, the last event in $<_{n+1}$ is a, and $\mathsf{tr}(a)$ is not complete. Hence, $\textsc{ComputeReorderings}(h_{n+1}) = \emptyset$; so $h_{n+1} = h_n \oplus_j a$. Then, as $\leq_{n+1}$ is an extension of $\leq_n$ and h_n only has one pending transaction, by our choice of $\textsc{next}$ function (see Section 3.5.1), a belongs to the only pending transaction. This shows that 1 holds.

  For proving that 2 holds for h_{n+1}, let $e \in \mathsf{P}, e' \in h_{n+1}$ s.t. $e <_{\mathsf{or}} e'$. As Property 2 holds in h_n and $\leq_{n+1}$ coincides with $\leq_n$ on events in h_n, if $e' \neq a$ or $e \in h_n$ the result immediately holds. Otherwise, if $e' = a$ and $e \notin h_n$, let us consider b the `begin` event in $\mathsf{tr}(a)$. By the definition of $\textsc{Next}$, $b \in h_n$. Thus, as Property 2 holds in h_n, we deduce that $e \not\leq_n b$; where b is the `begin` event in $\mathsf{tr}(a)$. Hence, condition 2b holds in h_n; so it holds in h_{n+1}.

- $\underline{a \text{ is not a } \texttt{begin} \text{ nor a } \texttt{read} \text{ event, and } h_{n+1} = h_n \oplus_j a}$: In this case, as $\leq_{n+1}$ is an extension of $\leq_n$ and h_n only has one pending transaction, by our choice of $\textsc{next}$ function (see Section 3.5.1), a belongs to the only pending transaction. This shows that 1 holds.

  For proving that 2 holds for h_{n+1}, let $e \in \mathsf{P}, e' \in h_{n+1}$ s.t. $e <_{\mathsf{or}} e'$. As Property 2 holds in h_n and $\leq_{n+1}$ coincides with $\leq_n$ on events in h_n, if $e' \neq a$ or $e \in h_n$ the result immediately holds. Otherwise, if $e' = a$ and $e \notin h_n$, let us consider b the `begin` event in $\mathsf{tr}(a)$. By the definition of $\textsc{Next}$, $b \in h_n$. Thus, as Property 2 holds in h_n, we deduce that $e \not\leq_n b$; where b is the `begin` event in $\mathsf{tr}(a)$. Hence, condition 2b holds in h_n for events b and e; so it also holds in h_{n+1} for events b and e.

- $\underline{a \text{ is not a } \texttt{begin} \text{ nor a } \texttt{read} \text{ event, and } h_{n+1} \neq h_n \oplus_j a}$: In this case, we observe that $(h_{n+1}, \mathsf{locals}_h) = \textsc{Swap}((h_n \oplus_j a, \leq_n), r, t, \mathsf{locals}'_n)$; where r, t are a read event and a transaction s.t. $(r, t) \in \textsc{ComputeReorderings}(h_n)$ and $\mathsf{locals}'_n = \mathsf{locals}_n[e \mapsto \gamma]$. Hence, $\textsc{ComputeReorderings}(h_{n+1}) \neq \emptyset$; so $a = \texttt{commit}$. Thus, $h_n \oplus_j a$ has no pending transactions. By the definition of $\textsc{Swap}$ (see Section 3.5.2), we conclude that h_{n+1} contains exactly one pending transaction, and it is the last one w.r.t. $\leq_{n+1}$; so 1 holds.

  For proving that 2 holds in h_{n+1}, let $e \in \mathsf{P}, e' \in h_{n+1}$ be two events s.t. $e \leq_{\mathsf{or}} e'$. For clarity, let so and wr (respectively so and wr) be the session order and write-read of h_{n+1} (resp. h_n). Three sub-cases arise:

  - $\underline{e \leq_{n+1} e'}$: In this case, Property 2a immediately holds in h_{n+1}.
  - $\underline{e' <_{n+1} e \text{ and } e' <_n e}$: In this case, as $e' <_n e$, we deduce that $e' \in h_n$ and $e' \in h_{n+1}$. As Property 2 holds in h_n but $e' \leq_n e$, Property 2b holds in h_n for events e and e'. Let (e'', t'') be an event and a transaction such property in h_n for events e and e'. As $\textsc{swapped}(h_n, e'')$ holds, by $\textsc{Optimality}$'s definition, $e'' \in h_{n+1}$ and $e'' \neq r$. Hence, as $e'' \neq r$, we deduce that $(\mathsf{tr}(e'), t) \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t, e'') \in \mathsf{wr}$ and $e'' \leq_{n+1} e$. Moreover, as $\mathsf{wr}^{-1}(e'') \in h_n$, by $\textsc{Optimality}$'s definition, we deduce that it is different from r and it also belongs to h_{n+1}. To sum up, $\textsc{swapped}(h_{n+1}, e'')$ holds as well; so Property 2 holds in h_{n+1}.

- $\underline{e' <_{n+1} e \text{ and } e \leq_n e'}$: In this case, we show that r and t are the event and transaction respectively for which Property 2b holds in h_{n+1} for events e and e'.
  - $\underline{r \leq_{n+1} e}$: As $e' <_{n+1} e$ and $e \leq_n e'$, we deduce that both $e, e' \in h_n$. Moreover, by the construction of h_{n+1} using SWAP function, we deduce that either $e \notin h_{n+1}$ or $e \in \mathsf{tr}(r)$; so $r \leq_{n+1} e$.
  - $\underline{(\mathsf{tr}(e'), t) \in (\mathsf{so} \cup \mathsf{wr})^*}$: We show that $r <_n e'$. On one hand, if $e \notin \mathsf{tr}(r)$, $e \notin h_n$. In such case, as $e' <_{n+1} e$ and $e \leq_n e'$, we conclude that $r <_n e$; so $r <_n e \leq_n e'$. On the other hand, if $e \in \mathsf{tr}(r)$, as $e \leq_n e'$ and $e' <_{n+1} e$, we deduce by Lemma 3.5.8 that $e' \notin \mathsf{tr}(r)$. In particular, we obtain $r <_n e'$. Then, as regardless of whether $e \in \mathsf{tr}(r)$ or not, $r \leq_n e'$, and as $e' \in h_{n+1}$, we conclude that $(\mathsf{tr}(e'), t) \in (\mathsf{so}' \cup \mathsf{wr}')^*$.
  - $\underline{(t, r) \in \mathsf{wr}}$: This is immediate by the definition of SWAP.
  - $\underline{r \leq_{\mathsf{or}} e}$: If $e \in \mathsf{tr}(r)$, the result immediately holds. Otherwise, if $e \notin \mathsf{tr}(r)$, we prove the result by by contrapositive, assuming that $r >_{\mathsf{or}} e$ and reaching a contradiction. First, as $e' <_{n+1} e$ and $e \leq_n e'$, we deduce that $e \notin h_{n+1}$ but $e \in h_n$. By SWAP's definition, we observe that $r <_n e$ and $(\mathsf{tr}(e), t) \notin (\mathsf{so} \cup \mathsf{wr})^*$. In such case, as h_n is oracle-respectful, we deduce that Property 2b holds in h_n for events r and e. Let e'' and t'' be an event and a transaction resp. witnessing it. In such case, $(\mathsf{tr}(r), t) \in (\mathsf{so}' \cup \mathsf{wr}')^*$, $(t, e'') \in \mathsf{wr}'$, $e'' \leq_n e$ and SWAPPED$((h_n, \leq_n), e'')$. As $(\mathsf{tr}(r), \mathsf{tr}(e'')) \in (\mathsf{so}' \cup \mathsf{wr}')^*$, by Lemma 3.5.8, we deduce that $r \leq_n e''$. Hence, as $r \leq_n e''$ and SWAPPED$((h_n, \leq_n), e'')$ holds, by OPTIMALITY and SWAP's definition, $(\mathsf{tr}(e''), t) \in (\mathsf{so}' \cup \mathsf{wr}')^*$. Altogether, we deduce that $(\mathsf{tr}(r), t) \in (\mathsf{so}' \cup \mathsf{wr}')^*$. However, this contradicts that $(r, t) \in$ COMPUTEREORDERINGS(h_n). In conclusion, the initial hypothesis, $r >_{\mathsf{or}} e$, is false; so $r \leq_{\mathsf{or}} e$.
  - $\underline{\text{SWAPPED}(h_{n+1}, r) \text{ holds}}$: Thanks to the SWAP's definition, we know that $t <_{n+1} r$ and that r is the only read event in $\mathsf{tr}(r)$ reading from t. Moreover, as r is the last event w.r.t. $<_{n+1}$ and t is the second to last transaction in h_{n+1} w.r.t. $<_{n+1}$, for proving that SWAPPED(h_{n+1}, r) holds it suffices to show that $t >_{\mathsf{or}} r$. We reason by contrapositive, assuming that $t \leq_{\mathsf{or}} r$ and reaching a contradiction.
    Let b be the `begin` event in t. As or respects the program order, $b \leq_{\mathsf{or}} r$. In such case, as h_n is oracle-respectful and $r <_n b$, we deduce that Property 2b holds in h_n for r and b. Let e'', t'' be an event and a transaction resp. satisfying the Property in h_n. In such case, $(\mathsf{tr}(r), t'') \in (\mathsf{so}' \cup \mathsf{wr}')^*$, $(t'', e'') \in \mathsf{wr}'$, $e'' \leq_n b$ and SWAPPED$((h_n, <_n), e'')$ holds. By Lemma 3.5.8, this implies that $r <_n e''$. Hence, by the definition of OPTIMALITY, as SWAPPED$((h_n, <_n), e'')$ holds, we deduce that $(\mathsf{tr}(e''), t) \in (\mathsf{so}' \cup \mathsf{wr}')^*$. Therefore, $(\mathsf{tr}(r), t) \in (\mathsf{so}' \cup \mathsf{wr}')$. However, this is impossible as $(r, t) \in$ COMPUTEREORDERINGS$((h_n, <_n))$; so the initial hypothesis, that $t \leq_{\mathsf{or}} r$ is false; so $r <_{\mathsf{or}} t$.

□

The following result is just an observation obtained during Lemma 3.5.9.

Corollary 3.5.10. *Let h be a reachable history, p a computable path leading to h and r be an event in h. The predicate* $\textsc{swapped}(h, r)$ *holds iff there exists a pair of histories and local mappings* $(h_1, \mathsf{locals}_1), (h_2, \mathsf{locals}_2)$ *in p s.t.* $(h_2, \mathsf{locals}_2) = \textsc{swap}(h_1 \oplus_j e, r, t, \mathsf{locals}')$*; where* $(j, e, \gamma) = \textsc{Next}(\mathsf{P}, h_1, \mathsf{locals}_1)$ *and* $\mathsf{locals}' = \mathsf{locals}_1[e \mapsto \gamma]$.

Lemma 3.5.11. *For any reachable history $h_<$, $\leq^h \equiv \leq_h$.*

Proof. As $h_<$ is reachable, there exists a computable path p whose last transaction is $h_<$. We prove the result by induction on the histories in such path. The base case, when the history is h_0, is immediate as h_0 only contains one transaction, `init`. Let us thus assume that the result holds for the pair $(h_n, \texttt{local}_n)$, and let us prove it for the pair $(h_{n+1}, \mathsf{locals}_{n+1})$. We denote by $\leq_n$ and $\leq_{n+1}$ to the order of h_n and h_{n+1} respectively. Also, we denote by $\leq^n$ and $\leq^{n+1}$ to the canonical order over h_n and h_{n+1} respectively. By hypothesis, h_n is oracle-respectful. We distinguish several cases, depending on the relation between h_{n+1} and the event e s.t. $(j, e, \gamma) = \textsc{next}(\mathsf{P}, h_n, \mathsf{locals}_n)$. For clarity, let so and wr (respectively so and wr) be the session order and write-read of h_{n+1} (resp. h_n).

- $\underline{h_{n+1} = h_n \oplus_j e \text{ and } e = \texttt{begin}}$: In this case, as $e = \min_{\mathsf{or}} \mathsf{P} \setminus h_n$, (1) $\mathsf{tr}(e)$ is the last transaction in h_{n+1} w.r.t. $\leq_{n+1}$ and (2) the relative order between other transactions in h_n coincide w.r.t. $\leq_n$. Also, as $\mathsf{tr}(e)$ is $(\mathsf{so} \cup \mathsf{wr})^*$-maximal and $e = \min_{\mathsf{or}} \mathsf{P} \setminus h_n$, (1) $\mathsf{tr}(e)$ is the last transaction in h_{n+1} w.r.t. $\leq^{n+1}$ and (2) the relative order between other transactions in h_n coincide w.r.t. $\leq^n$. Altogether, we conclude that $\leq_{n+1} = \leq^{n+1}$.

- $\underline{h_{n+1} = h_n \oplus_j e \text{ and } e \neq \texttt{begin}}$: Let b be the `begin` event in $\mathsf{tr}(e)$. Then, as $\texttt{begin} <_{\mathsf{or}} e$ and $h_{n+1} = h_n \oplus_j e$, $\leq^{n+1} = \leq^n$. By induction hypothesis, $\leq^n = \leq_n$. Moreover, by Lemma 3.5.9, as $e \neq \texttt{begin}$, $\leq_n = \leq_{n+1}$. Altogether, $\leq^{n+1} = \leq_{n+1}$.

- $\underline{h_{n+1} = h_n \oplus_j e \oplus \mathsf{wr}(t, e)\text{; for some } t \in \textsc{ValidWrites}(h_n)}$: This case is identical to the previous one.

- $\underline{h_{n+1} \neq h_n \oplus_j e \text{ and } h_{n+1} \neq h_n \oplus_j e \oplus \mathsf{wr}(t, e)\text{; for every } t \in \textsc{ValidWrites}(h_n):}$ In this case, $(h_{n+1}, \mathsf{locals}_{n+1}) = \textsc{Swap}(h_n \oplus_j e, r, t, \mathsf{locals}'_n)$; where r, t are a read event and a transaction s.t. $(r, t) \in \textsc{ComputeReorderings}(h_n)$ and $\mathsf{locals}'_n = \mathsf{locals}_n[e \mapsto \gamma]$.First, we analyze $\leq_{n+1}$ and $\leq^{n+1}$ for transactions in h_{n+1} different from $\mathsf{tr}(r)$. By $\textsc{Swap}$'s definition, for every transaction $t' \in h_{n+1}$ s.t. $t' \neq \mathsf{tr}(r)$, their $(\mathsf{so} \cup \mathsf{wr})^*$-predecessors coincide with its $(\mathsf{so}' \cup \mathsf{wr}')^*$-predecessors. Hence, for every event e', $\min_{\mathsf{or}} \textsc{dep}(h_n, t', e') = \min_{\mathsf{or}} \textsc{dep}(h_{n+1}, t', e')$. Therefore, for every pair of distinct transactions $t_1, t_2 \in h_{n+1}$ different from $\mathsf{tr}(r)$, $t_1 \leq^n t_2$ iff $t_1 \leq^{n+1} t_2$. Also, by $\textsc{Swap}$'s definition, the only transaction where $\leq_n$ and $\leq_{n+1}$ do not coincide is $\mathsf{tr}(r)$. Applying the inductive hypothesis on h_n, we deduce that for every pair of distinct transactions $t_1, t_2 \in h_{n+1}$ different from $\mathsf{tr}(r)$, we deduce that $t_1 \leq^{n+1} t_2$ iff $t_1 \leq_{n+1} t_2$.

  Next, we analyze $\leq_{n+1}$ and $\leq^{n+1}$ for pairs of distinct transactions in h_{n+1} where one of them is $\mathsf{tr}(r)$. By $\textsc{Swap}$'s definition, for every transaction $t' \in h_{n+1}$ different from $\mathsf{tr}(r)$, $t' \leq_{n+1} \mathsf{tr}(r)$. We show that $t' \leq^{n+1} \mathsf{tr}(r)$. Clearly, if $(t', \mathsf{tr}(r)) \in$

$(\mathsf{so} \cup \mathsf{wr})^*$, $t' \leq^{n+1} \mathsf{tr}(r)$. Moreover, if $(t', \mathsf{tr}(r)) \not\in (\mathsf{so} \cup \mathsf{wr})^*$, as by SWAP's definition, $\mathsf{tr}(r)$ is $(\mathsf{so} \cup \mathsf{wr})^*$-maximal, $(\mathsf{tr}(r), t') \not\in (\mathsf{so} \cup \mathsf{wr})^*$. We show that in such case, MINIMALDEPENDENCY$(h_{n+1}, t', \mathsf{tr}(r), \perp)$ holds; so $t' \leq^{n+1} \mathsf{tr}(r)$ as well. Let $a = \min_{\mathsf{or}} \text{DEP}(h_{n+1}, \mathsf{tr}(r), \perp)$ and $a' = \min_{\mathsf{or}} \text{DEP}(h_{n+1}, t', \perp)$. We note that as $\mathsf{tr}(r)$ is $(\mathsf{so} \cup \mathsf{wr})^*$-maximal, $a \in \mathsf{tr}(r)$. Let also b be the `begin` event of t'. Then, on one hand, if $b <_{\mathsf{or}} a$, $a' <_{\mathsf{or}} a$; so MINIMALDEPENDENCY$(h_{n+1}, t', \mathsf{tr}(r), \perp)$ holds. On the other hand, if $a \leq_{\mathsf{or}} b$, as h_{n+1} is oracle-respectful (Lemma 3.5.9) and $t' \leq_{n+1} \mathsf{tr}(r)$, Property 2b holds in h_{n+1} for events a and b. Then, let e'' and t'' be an event and transaction respectively witnessing it. In such case, $e'' \leq_{\mathsf{or}} a$ and $e'' \in \text{DEP}(h_{n+1}, t', \perp)$; so $a' \leq_{\mathsf{or}} a$. Finally, as $a' \not\in \mathsf{tr}(r)$, $a' \neq a$; so we conclude that $a <_{\mathsf{or}} a$ and that MINIMALDEPENDENCY$(h_{n+1}, t', \mathsf{tr}(r), \perp)$ holds.

□

We deduce from Lemma 3.5.11 that regardless of the computable path that leads to a history, the final order between events will be the same; and it coincides with the canonical order. Hence, we can assume w.l.o.g. that every history is ordered by simply appending it its canonical order. The last result, Corollary 3.5.10 states that SWAPPED captures exactly if an event has been swapped via a SWAP condition.

Another instance of oracle-respectful histories are total histories ordered with the canonical order.

Lemma 3.5.12. *For any total history h, the ordered history $(h, \leq^h)$ is oracle-respectful.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a total history. First as h is total, Property 1 holds. Next, for showing that Property 2 holds, let t_1, t_2 be a pair of distinct transactions s.t. $t_1 \leq_{\mathsf{or}} t_2$ but $t_2 \leq^h t_1$, and let us prove that Property 2b holds.

On one hand, if $(t_2, t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$, by Lemma 3.5.5, we deduce that there exists an event r and a transaction t' s.t. $(t_2, t') \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t', r) \in \mathsf{wr}$, $(\mathsf{tr}(r), t_1) \in (\mathsf{so} \cup \mathsf{wr})^*$, $r \leq_{\mathsf{or}} t_2$ and $r \leq_{\mathsf{or}} t'$. W.l.o.g. we can assume that r is minimal event w.r.t. $\leq^h$. We observe that as $(t', r) \in \mathsf{wr}$, $t' \leq^h r$. Then, thanks to the minimality of r w.r.t. $\leq^h$, we conclude that SWAPPED(h, r) holds; so Property 2b holds.

On the other hand, if $(t_2, t_1) \not\in (\mathsf{so} \cup \cup \mathsf{wr})^*$, as $t_2 \leq^h t_1$, $(t_1, t_2) \not\in (\mathsf{so} \cup \mathsf{wr})^*$ and MINIMALDEPENDENCY$(h, t_2, t_1, \perp)$ holds. We define two sequence of events e_i^1 and e_i^2 and e_i^3 representing the recursive calls to DEP of t_1 and t_2. For each $j \in \{1, 2, 3\}$, we define $e_0^j = \perp$ and for each $i \in \mathbb{N}$, $e_{i+1}^j = \text{DEP}(h, t_j, e_i^j)$. Let n_0 be the maximum index s.t. $e_i^1 = e_i^2$. As $t_1 \neq t_2$, by Lemma 3.5.2, n_0 is well-defined. Moreover, as $t_2 \leq^h t_1$, there exists a transaction t' s.t. $(t_2, t') \in (\mathsf{so} \cup \mathsf{wr})^*$, $(t', e_{n_0+1}^2) \in \mathsf{wr}$ and $e_{n_0+1}^2 \leq_{\mathsf{or}} e_{n_0+1}^1$. By the definition of $e_{n_0+1}^1$, we deduce that $e_{n_0+1}^2 \leq_{\mathsf{or}} t_1$. We show that $e_{n_0+1}^2$ is exactly the searched event.

Clearly, by the definition DEP, there exists a transaction t'' s.t. $(t_2, t'') \in (\mathsf{so} \cup \mathsf{wr})^*$ and $(t'', e_{n_0+1}^2) \in \mathsf{wr}$. Also, as $e_{n_0+1}^2 \leq_{\mathsf{or}} e_{n_0+1}^1$, we deduce that $e_{n_0+1}^2 \leq_{\mathsf{or}} t_1$. Moreover, by the definition of $e_{n_0+1}^2$, we deduce that for every $i, 0 \leq i \leq n_0$, there exists a transaction t_i s.t. $(e_{n_0+1}^2, t_i) \in (\mathsf{so} \cup \mathsf{wr})^*$ and $(t_i, e_i^2) \in \mathsf{wr}$. Therefore, MINIMALDEPENDENCY$(h, \mathsf{tr}(e_{n_0}^2), t_1, \perp)$ holds; and hence, $e \leq^h t_1$. Finally, $e_{n_0+1}^2$'s minimality w.r.t. or as well as the fact that $t_1 \leq_{\mathsf{or}} t_2$ ensures that SWAPPED$(h, e_{n_0+1}^2)$ holds.

□

3.5.4.3 Previous of a History

In this section, we construct the *previous* of a history, which in the case of reachable histories, it corresponds to the immediate predecessor in a computable path. This function shows that Swap is bijective; so computable paths are always unique.

Algorithm 4 Previous of a history

```
 1: procedure PREV($\mathsf{P}, h = (T, \mathsf{so}, \mathsf{wr})$)
 2:   $a \leftarrow \mathtt{last}(h)$
 3:   if $h = (\mathtt{init}, \emptyset, \emptyset)$ then return $h$
 4:   else if $\neg$SWAPPED$(h, a)$ then return $h \setminus a$
 5:   else
 6:     let $D = \{e \mid e \in \mathsf{P} \setminus (h \setminus \{a\}) \land e <_{\mathsf{or}} \mathsf{wr}^{-1}(a)\}$
 7:     return MAXCOMPLETION$(h \setminus \{a, \mathsf{tr}(\mathtt{last}(\mathsf{wr}^{-1}(a)))\}, D)$

 8: procedure MAXCOMPLETION($h = (T, \mathsf{so}, \mathsf{wr}), D$)
 9:   if $D \neq \emptyset$ then
10:     $e \leftarrow \min_{<_{\mathsf{or}}} D$
11:     if $\mathtt{op}(e) \neq \mathtt{read}$ then return MAXCOMPLETION$(h \oplus e, D \setminus \{e\})$
12:     else
13:       let $t$ s.t. $\mathsf{readLatest}_\iota(h \oplus e \oplus \mathsf{wr}(t, e), e,)$ holds
14:       return MAXCOMPLETION$(h \oplus e \oplus \mathsf{wr}(t, e), D \setminus \{e\})$
15:   elsereturn $h$
```

First, we show that oracle-respectfulness is preserved via PREV.

Lemma 3.5.13. *For every oracle-respectful history h,* PREV(h) *is also oracle-respectful.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history and $v =$ PREV(h) be its previous. We denote by $\leq^h$ and $\leq^v$ to their respectively canonical order; and we denote by T', so' and wr' to the transactions, session order and write-read relation of v respectively. Also, we denote by a to the last event in h. Three cases arise depending on which condition among 3, 4 and 7 holds.

- $\underline{h = (\mathtt{init}, \emptyset, \emptyset)}$: In this case, $v = h$ and the result immediately holds.
- $\underline{h \neq (\mathtt{init}, \emptyset, \emptyset) \text{ and } \neg\text{SWAPPED}(h, a) \text{ holds}}$: In this case, $v = h \setminus a$. As h is oracle-respectful, Property 1 holds in v. Also, as $\leq^v \subseteq \leq^h$ and a is not swapped, Property 2 holds in v; so v is oracle-respectful.
- $\underline{h \neq (\mathtt{init}, \emptyset, \emptyset) \text{ and } \text{SWAPPED}(h, a) \text{ holds}}$: In this case, the history v is defined as in line 7 of Section 3.5.4.3. Note that $a \neq \mathtt{commit}, \mathtt{abort}$, $\mathsf{tr}(a)$ must be the pending transaction of h. Moreover, as a is swapped, $\mathsf{tr}(a) \leq_{\mathsf{or}} \mathsf{wr}^{-1}(a)$. Hence, $\mathsf{tr}(a)$ is not pending

in v. By construction of v, v contains exactly one pending transaction: $\mathsf{tr}(\mathsf{wr}^{-1}(a))$. As $\mathtt{last}(v) = \mathsf{tr}(\mathsf{wr}^{-1}(a))$, we conclude that Property 1 holds.

For showing that Property 2 holds, let $e \in \mathsf{P}$, $e' \in v$ be a pair of events s.t. $e \leq_{\mathsf{or}} e'$ but $e' \leq^v e$; and let us show that Property 2b of Definition 3.5.7 holds. As $e \leq_{\mathsf{or}} e'$ but $e' \leq^v e$ Lemma 3.5.8, we deduce that there exists a transaction t'' and an event e'' s.t. $(\mathsf{tr}(e'), t'') \in (\mathsf{so}' \cup \mathsf{wr}')^*$, $(t'', e'') \in \mathsf{wr}'$, $t'' \leq^v e''$ and $e'' \leq_{\mathsf{or}} t''$. Choosing e'' minimal w.r.t. $\leq^v$, allow us to conclude that $e'' \leq^v e$ and $\textsc{swapped}(v, e)$ holds: every read event in $\mathsf{events}(h) \setminus \{a\}$ reads from the same transaction in h and in v. Moreover, by the definition of $\mathsf{readLatest}$, every transaction in $T' \setminus (T \setminus \mathsf{tr}(a))$ reads from causally-dependent transactions. Hence, the only swapped events in v are those swapped events in h different from a. Altogether, we deduce that $e'' \leq^v e$ and $\textsc{swapped}(v, e)$ holds.

□

Next, we have to prove that $\textsc{prev}$ is compatible with $\textsc{explore-ce}$.

Lemma 3.5.14. *For every isolation level ι that is causally-extensible and every history h that is oracle-respectful and consistent w.r.t. ι, if $\textsc{prev}(h)$ is reachable, then h is also reachable.*

Proof. Let us assume that $h \neq (\mathtt{init}, \emptyset, \emptyset)$, as otherwise the result immediately holds, let $v = \textsc{prev}(h)$ be the previous of h and p be a computable path that leads to v. We show that from p we reach h in one $\textsc{explore-ce}$ call. For that, let $a = \mathtt{last}(h)$ be the last event of h, let locals_v be the local variables of v in the path p and let j, e and γ be respectively the session index, event and the state such that $(j, e, \gamma) = \textsc{Next}(\mathsf{P}, v, \mathsf{locals}_v)$. Let us also denote by so (resp. so') and wr (resp. wro') to the session order and write-read relations of h (resp. of v).

On one hand, if $\neg\textsc{swapped}(h, a)$ holds, by the choice of $\textsc{Next}$, $e = a$ and $j = \mathsf{ses}(\mathsf{tr}(a))$. We observe that if a is not a read, $h = v \oplus a$; while if it is, $h = v \oplus a \oplus \mathsf{wr}(t, a)$; for some transaction $t \in v$. Note that as h is consistent w.r.t. ι, in the latter case, $t \in \textsc{valid}_\iota(h, a)$. Hence, h is reachable by the computable path that, after p, contains the tuple $(h, \mathsf{locals}_v[a \mapsto \gamma])$.

On the other hand, if $\textsc{swapped}(h, a)$ holds, we show first that v is well-defined. For that, it suffices to show that for every read event $r \in \mathsf{events}(v) \setminus (\mathsf{events}(h) \setminus \{a\})$, the transaction r reads from exists (line 13). We observe that h and all its recursive calls to $\textsc{maxCompletion}$ function have at most pending transaction at a time, which is $(\mathsf{so} \cup \mathsf{wr})^*$-maximal. Therefore, as ι is causally-extensible by hypothesis, such transaction exists. Moreover, any such read event is not swapped in v.

Then, to show that v is reachable, we simply show that the pair $(a, t) \in \textsc{ComputeReorderings}(v \oplus_j e)$; where t is the transaction s.t. $(t, a) \in \mathsf{wr}$. As a is swapped, $a \leq_{\mathsf{or}} t$. As $(t, a) \in \mathsf{wr}$ but $a \leq_{\mathsf{or}} t$, we deduce that they do belong to different sessions. Moreover, by construction of v, we deduce that the event $e = \mathtt{last}(\mathsf{tr}(\mathsf{wr}^{-1}(a)))$. Hence, by Section 3.5.4.3, $a \leq_v t$ and $(\mathsf{tr}(a), t) \notin (\mathsf{so}' \cup \mathsf{wr}')^*$. In addition, by $\textsc{prev}$'s definition, we know that $e = \mathtt{last}(\mathsf{tr}(\mathsf{wr}^{-1}(a)))$. Therefore, $(a, t) \in \textsc{ComputeReorderings}(v \oplus_j e)$. In conclusion, h is reachable by the computable path that, after p, contains the tuple $(\textsc{Swap}(v, r, t, \mathsf{locals}_h), \mathsf{locals}_h)$; where $\mathsf{locals}_h = \mathsf{locals}_v[a \mapsto \gamma]$. □

As an observation of the proof of Lemma 3.5.14, we deduce the following result.

Corollary 3.5.15. *Let ι be an isolation level, and let h be an oracle-respectful consistent w.r.t. ι history whose previous history is reachable. If* $\textsc{swapped}(h, a)$ *holds, then h is obtained by a swap operation from* $\textsc{prev}(h)$*; where $a = \texttt{last}(h)$.*

3.5.4.4 Completeness of Algorithm EXPLORE-CE

For concluding the completeness of algorithm EXPLORE-CE, we prove an invariant relation between an oracle-respectful history and its previous that will allow us to deduce completeness. The invariant intuitively says that a history (1) has more events or more swapped events than its predecessor and (2) contains all swapped events of its predecessor. Formally, let h be a history, $v = \textsc{prev}(h)$ and $a = \texttt{last}(h)$. The invariant of h and v, $I(h, v)$, is defined as follows:

$$I(h, v) = (\mathsf{S}_e^v = \mathsf{S}_e^h \setminus \{a\}) \;\vee\; (\mathsf{S}_e^v = \mathsf{S}_e^h \;\wedge\; \mathsf{events}(v) = \mathsf{events}(h) \setminus \{a\}) \tag{3.4}$$

where $\mathsf{S}_e^h = \{e \in \mathsf{events}(h) \mid \textsc{swapped}(h, e)\}$ and $\mathsf{S}_e^v = \{e \in \mathsf{events}(v) \mid \textsc{swapped}(v, e)\}$.

Lemma 3.5.16. *Let h be a oracle-respectful history different from* $(\texttt{init}, \emptyset, \emptyset)$. *The invariant* $I(h, \textsc{prev}(h))$ *holds.*

Proof. For proving the result, we simply analyze PREV function. Let a be the last event in h and let $v = \textsc{prev}(h)$. If $\neg\textsc{swapped}(h, a)$ holds, $v = h \setminus a$. Hence, $\{e \in \mathsf{events}(v) \mid \textsc{swapped}(v, e)\} = \{e \in \mathsf{events}(h) \mid \textsc{swapped}(h, e)\}$ and $\mathsf{events}(v) = \mathsf{events}(h) \setminus \{a\}$ hold. Otherwise, we observe that as h is oracle-respectful, v is oracle-respectful as well (Lemma 3.5.13). Therefore, (1) if $e \neq a$ is swapped in h, it is also swapped in v and (2) if $e \in \mathsf{events}(h) \setminus \mathsf{events}(v)$, then e is not swapped (see details in Lemma 3.5.13's proof). Hence, $\{e \in \mathsf{events}(v) \mid \textsc{swapped}(v, e)\} = \{e \in \mathsf{events}(h) \mid \textsc{swapped}(h, e)\} \setminus \{a\}$ holds. Altogether, we conclude that $I(h, \textsc{prev}(h))$ holds.

□

We prove using the aforementioned invariant that after applying a finite number of times the function PREV we reach the initial history.

Lemma 3.5.17. *For every oracle-respectful history h there exists some $k_h \in \mathbb{N}$ such that* $\textsc{prev}^{k_h}(h) = (\texttt{init}, \emptyset, \emptyset)$.

Proof. This result is immediate consequence of Lemma 3.5.16. Let $s(h)$ be the number of swapped events in h, and let us prove the lemma by induction on $(s(h), \mathsf{events}(h))$. The base case, $s(h) = 0$ and $\mathsf{events}(h) = 1$ is trivial, as in this case h is the initial history. For the inductive case, let us assume that for every history h such that $s(h) < n$ or $s(h) = n$ and $\mathsf{events}(h) < m$ there exists such k_h; and let us prove the result for a history h' s.t. $s(h') = n$ and $\mathsf{events}(h') = m$. Let $v = \textsc{prev}(h')$ be the previous of h'. As Lemma 3.5.16 holds, we deduce that either $s(v) < s(h') = n$ or $s(v) = s(v') = n$ and $\mathsf{events}(v) = \mathsf{events}(h') - 1 < m$. Either way, by inductive hypothesis, we deduce that there exists k_v s.t. $\textsc{prev}^{k_v}(v) = (\texttt{init}, \emptyset, \emptyset)$. Therefore, $k_{h'} = k_v + 1$ satisfies that $\textsc{prev}^{k_{h'}}(h') = (\texttt{init}, \emptyset, \emptyset)$. □

Lemma 3.5.18. *Let ι be a causally-extensible isolation level. For every consistent w.r.t. ι oracle-respectful history h exists $k \in \mathbb{N}$ and a computable path of length k, $p = \{(h_n, \mathsf{locals}_n)\}_{n=0}^{k}$, s.t. $h_0 = (\mathtt{init}, \emptyset, \emptyset)$, $h_k = h$ and every history h_n is consistent w.r.t. ι.*

Proof. Let k be the minimum integer such that $\textsc{prev}^k(h) = (\mathtt{init}, \emptyset, \emptyset)$, which exists thanks to Lemma 3.5.17 and let $H_h = \{\textsc{prev}^{k-n}(h)\}_{n=0}^{k}$ be the sequence of histories. By construction, H_h is a sequence of histories s.t. $h_0 = \textsc{prev}^k(h) = (\mathtt{init}, \emptyset, \emptyset)$, $h_k = \textsc{prev}^0(h) = h$. Also, as h is oracle-respectful, by Lemma 3.5.13, every history in H_h is oracle-respectful. Then, we observe that $h_0 = (\mathtt{init}, \emptyset, \emptyset)$ is reachable and consistent w.r.t. ι. Let locals_0 be the local state of h_0. Then, by Lemma 3.5.14, we conclude that every history h_n is reachable (on some state locals_n). Hence, the path $p = \{(h_n, \mathsf{locals}_n)\}_{n=0}^{k}$ satisfies the result.

□

Theorem 3.5.19. *Let ι be a causally-extensible isolation level. The algorithm* EXPLORE-CE *is complete.*

Proof. By Lemma 3.5.12, any consistent w.r.t. ι total history h is oracle-respectful. As a consequence of Lemma 3.5.18, there exist a sequence of reachable histories which h belongs to; so in particular, h is reachable. □

3.5.4.5 Strong Optimality of Algorithm EXPLORE-CE

The proof of strong optimality relies on the notions of oracle-respectfulness (Section 3.5.4.2) and canonical order of a history (Section 3.5.4.1). As EXPLORE-CE does not engage in fruitless explorations, it suffices to prove optimality. In particular, we prove that the computable path obtained in Lemma 3.5.18 is unique.

Theorem 3.5.20. *Let ι be a causally-extensible isolation level. The algorithm* EXPLORE-CE *is strongly optimal.*

Proof. Let us prove that for every reachable history there is only a computable path that leads to it from the initial history. By contrapositive, let us assume that there exists a history $h = (T, \mathsf{so}, \mathsf{wr})$ that is reached by two computable paths, p_1 and p_2. By Lemma 3.5.11, we know that $\leq_h \equiv \leq^h$. However, $\leq^h$ is an order that does not depend on the computable path that leads to h; so neither does $\leq_h$. Therefore, we can assume without loss of generality that h is a history with minimal value of $s(h) = |\{e \in \mathsf{events}(h) \mid \textsc{swapped}(h, e)\}|$ and in case of tie, that is minimal with respect $|\mathsf{events}(h)|$.

As shown in the proof of Lemma 3.5.18, if h is reachable, then $\textsc{prev}(h)$ is reachable as well; so we can assume w.l.o.g. that the predecessor of h in p_1 is $h_1 = \textsc{prev}h$. We first show that $h_1 = h_2$ and then conclude that $p_1 = p_2$; where h_2 is the predecessor of h in p_2. Let in the following $\mathsf{locals}_1, \mathsf{locals}_2$ be the local state of h_1 and h_2 resp. in p_1 and p_2 resp., and let a be the last event in h. On one hand, if a is not a swapped read event, by the definition of $\textsc{next}$ function $h_2 = h \setminus \{a\} = h_1$.

On the other hand, if a is swapped event and the last event in h, by Corollary 3.5.10, we deduce that $h = \textsc{Swap}(h_1 \oplus_{j_1} e_1, a, t)$; where t is the transaction s.t. $(t, a) \in \mathsf{wr}$ and

j_1, e_1 and γ_1 are respectively the session, the event and the local state s.t. $(j_1, e_1, \gamma_1) = \text{NEXT}(\mathsf{P}, h_1, \mathsf{locals}_1)$. The same observation can be done for history h_2, so we deduce that $h = \text{SWAP}(h_2 \oplus_{j_1} e_2, a, t)$; where j', e' and γ' are respectively the session, the event and the local state s.t. $(j_2, e_2, \gamma_2) = \text{NEXT}(\mathsf{P}, h_2, \mathsf{locals}_2)$. By COMPUTEREORDERINGS's definition, we deduce that $j_1 = j_2$ and $e_1 = e_2$. However, as they are both reachable, by Lemma 3.5.9, they are oracle-respectful. Hence, h_1 and h_2 must contain the same events. Moreover, by OPTIMALITY's definition, we conclude that h_1 and h_2 not only contain the same events, but also that their read events must read from the same transactions. Altogether, we deduce that $h_1 = h_2$.

Finally, as by Lemma 3.5.16, $I(h, \text{PREV}(h))$ holds, we apply the inductive hypothesis to $h_1 = \text{PREV}(h)$. As there is only one computable path leading to h_1 and NEXT is deterministic, we conclude that there is only one computable path leading to h. □

3.6 Swapping-Based Model Checking Algorithms for Snapshot Isolation and Serializability

For EXPLORE-CE, *not* engaging in fruitless explorations is a direct consequence of causal extensibility (of the isolation level). However, isolation levels such as `SI` and `SER` are *not* causally extensible (see Section 3.3.2). Therefore, whether there exists another implementation of EXPLORE that can ensure strong optimality, with soundness and completeness w.r.t. ι for SI or SER remains open. We answer this question in the negative, and as a result, propose an SMC algorithm that extends EXPLORE-CE by just filtering histories before outputting to be consistent with respect to `SI` or `SER`.

Theorem 3.6.1. *If ι is Snapshot Isolation or Serializability, there exists no* EXPLORE *algorithm that is sound and complete w.r.t. ι and strongly optimal.*

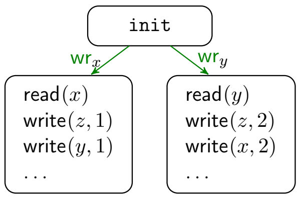
Proof. We consider the program in Figure 3.13a, and show that any concrete instance of the EXPLORE function in Algorithm 1 *can not be both* complete w.r.t. ι and strongly optimal. This program contains two transactions, where only the first three instructions in each transaction are important. We show that if EXPLORE is complete w.r.t. ι, then it will necessarily be called recursively on a history h like in Figure 3.13b which does not satisfy ι, thereby violating strong optimality. In the history h, both *Snapshot Isolation* and *Serializability* forbid the two reads reading initial values while the writes following them are also executed (committed). A diagram of the proof can be seen in Figure 3.14.

Assuming that the function NEXT is not itself blocking (which would violate strong optimality), the EXPLORE will be called recursively on *exactly one* of the two histories in Figure 3.13c, depending on which of the two reads is returned first by NEXT. We will continue our discussion with the history h_1 on the top of Figure 3.13c. The other case is similar (symmetric).

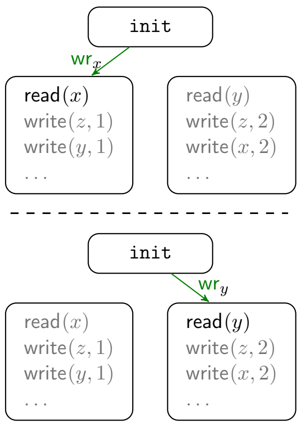
From h_1, depending on order defined by NEXT between $\texttt{write}(z, 1)$ and $\texttt{read}(y)$, EXPLORE can be called recursively either on h_{11} in Figure 3.13f or on h_2 in Figure 3.13d. Analogously, from h_2 two alternatives arise depending on the order defined by NEXT between $\texttt{read}(y)$ and the rest of events in the left transaction: exploring h_{21} in Figure 3.13g if $\texttt{read}(y)$ is added

```
begin;             || begin;
a = read(x);       || b = read(y);
write(z,1);        || write(z,2);
write(y,1);        || write(x,2);
...                || ...
commit             || commit
```

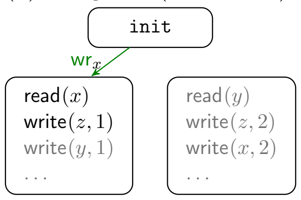
(a) Program (2 sessions).


init
wr_x
wr_y
read(x) write(z,1) write(y,1) ...
read(y) write(z,2) write(x,2) ...

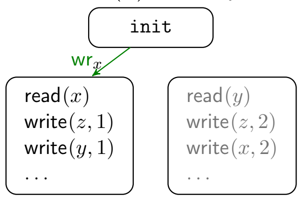
(b) History h.


init
wr_x
read(x) write(z,1) write(y,1) ...
read(y) write(z,2) write(x,2) ...
init
wr_y
read(x) write(z,1) write(y,1) ...
read(y) write(z,2) write(x,2) ...

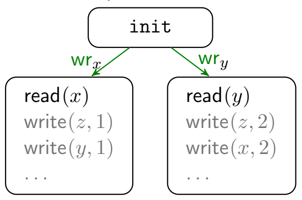
(c) Two histories. The top history is called h_1.


init
wr_x
read(x) write(z,1) write(y,1) ...
read(y) write(z,2) write(x,2) ...

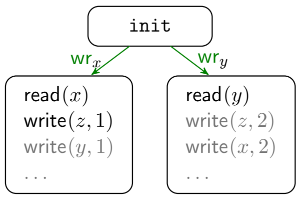
(d) History h_2.


init
wr_x
read(x) write(z,1) write(y,1) ...
read(y) write(z,2) write(x,2) ...

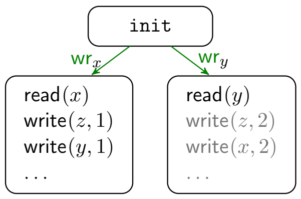
(e) History h_3.


init
wr_x
wr_y
read(x) write(z,1) write(y,1) ...
read(y) write(z,2) write(x,2) ...

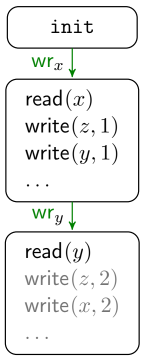
(f) History h_{11}.


init
wr_x
wr_y
read(x) write(z,1) write(y,1) ...
read(y) write(z,2) write(x,2) ...

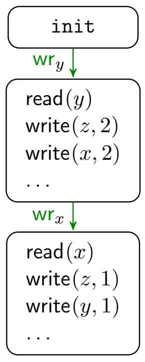
(g) History h_{21}.


init
wr_x
wr_y
read(x) write(z,1) write(y,1) ...
read(y) write(z,2) write(x,2) ...

(h) History h_{31}.


init
wr_x
read(x) write(z,1) write(y,1) ...
wr_y
read(y) write(z,2) write(x,2) ...

(i) History h_{32}.


init
wr_y
read(y) write(z,2) write(x,2) ...
wr_x
read(x) write(z,1) write(y,1) ...

(j) History $\hat{h}$.

Figure 3.13: A program and some partial histories. Events in grey are not yet added to the history. For h_3, h_{31} and h_{32}, the number of events that follow $\mathtt{write}(y, 1)$ and $\mathtt{write}(x, 2)$ is not important (we use black . . . to signify that).

before $\mathtt{write}(y, 1)$ or h_3 in Figure 3.13e otherwise. Thus, from h_3 two alternatives arise when added $\mathtt{read}(y)$ depending on where it reads from: h_{31} in Figure 3.13h if it reads from `init` and h_{32} in Figure 3.13i if it reads from the left transaction.

However, from histories h_{11}, h_{21} or h_{31} EXPLORE will necessarily be called recursively on a history h like in Figure 3.13b which does not satisfy ι, thereby violating strong optimality: EXPLORE always explore branches that enlarge the current history. Thus, any EXPLORE implementation that is strong optimal should only explore h_{32}. In such case, by the restrictions on the SWAP function (defined in Section 3.4), any extension of h_{32} does not allow to explore the history $\hat{h}$ in Figure 3.13e where $\texttt{read}(x)$ reads from $\texttt{write}(x, 2)$: any outcome of a re-ordering between two contiguous subsequences α and β must be prefix of such extension when the events in α are taken out. In particular, for any extension h' of h_{32} and pair of contiguous sequences α, β such that $h' \setminus \alpha$ is a prefix of h', if an event from the second transaction belongs to β, $\texttt{read}(y)$ must also be in β. Therefore, $\texttt{write}(x, 2)$ must be in β as it is $\mathsf{wr}^{-1}(\texttt{read}(y))$. Hence, $\texttt{read}(x)$ must also be in β. Analogously, if $\texttt{read}(x)$ belongs to β, $\texttt{init}$ belongs to it. Altogether, if β contains any element, then α must be empty; so no swaps can be produced from h_{32}. To conclude, in this case EXPLORE violates completeness w.r.t. ι.

□

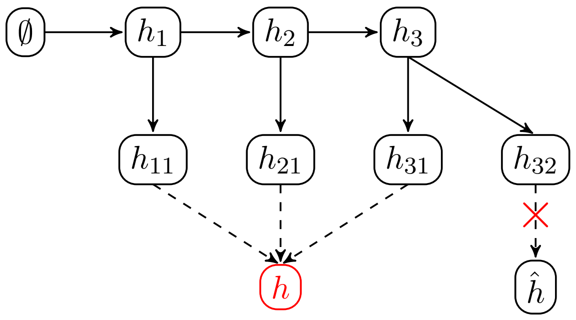


Figure 3.14: Summary of all possible execution paths from EXPLORE. Black arrows represent alternative explored options depending on NEXT while dashed arrows are mandatory visited histories from such state.

Given this negative result, we define an implementation of EXPLORE for an isolation level $\iota \in \{\texttt{SI}, \texttt{SER}\}$ that ensures optimality instead of strong optimality, along with soundness, completeness, and polynomial space bound. Thus, let EXPLORE-CE(ι_0) be an instance of EXPLORE-CE parametrized by $\iota_0 \in \{\texttt{RC}, \texttt{RA}, \texttt{TCC}\}$. We define an implementation of EXPLORE for ι, denoted by EXPLORE-CE$^*(\iota_0, \iota)$, which is exactly EXPLORE-CE(ι_0) except that instead of VALID(h) ::= true, it uses

$$\text{VALID}(h) \quad := \quad h \text{ satisfies } \iota$$

EXPLORE-CE$^*(\iota_0, \iota)$ enumerates exactly the same histories as EXPLORE-CE(ι_0) except that it outputs only histories consistent with ι. The following is a direct consequence of Theorem 3.5.1.

Corollary 3.6.2. *For any isolation levels ι_0 and ι such that ι_0 is prefix-closed and causally extensible, and ι_0 is weaker than ι,* EXPLORE-CE$^*(\iota_0, \iota)$ *is sound and complete w.r.t. ι, optimal, and employs polynomial space.*

3.7 Experimental Evaluation

We evaluate an implementation of EXPLORE-CE and EXPLORE-CE* in the context of the Java Pathfinder (JPF) [103] model checker for Java concurrent programs. As benchmark, we use bounded-size client programs of a number of database-backed applications drawn from the literature. The experiments were performed on an Apple M1 with 8 cores and 16 GB of RAM.

3.7.1 Implementation

We implemented our algorithms as an extension of the `DFSearch` class in JPF. For performance reasons, we implemented an iterative version of these algorithms where roughly, inputs to recursive calls are maintained as a collection of histories instead of relying on the call stack. For checking consistency of a history with a given isolation level, we implemented the algorithms proposed by [29].

Our tool takes as input a Java program and isolation levels as parameters. We assume that the program uses a fixed API for interacting with the database, similar to a key-value store interface. This API consists of specific methods for starting/ending a transaction, and reading/writing a global variable. The fixed API is required for being able to maintain the database state separately from the JVM state (the state of the Java program) and update the current history in each database access. This relies on a mechanism for "transferring" values read from the database state to the JVM state.

3.7.2 Benchmark

We consider a set of benchmarks inspired by real-world applications and evaluate them under different types of client programs and isolation levels.

Shopping Cart [97] allows users to add, get and remove items from their shopping cart and modify the quantities of the items present in the cart.

Twitter [51] allows users to follow other users, publish tweets and get their followers, tweets and tweets published by other followers.

Courseware [87] manages the enrollment of students in courses in an institution. It allows to open, close and delete courses, enroll students and get all enrollments. One student can only enroll to a course if it is open and its capacity has not reached a fixed limit.

Wikipedia [51] allows users to get the content of a page (registered or not), add or remove pages to their watching list and update pages.

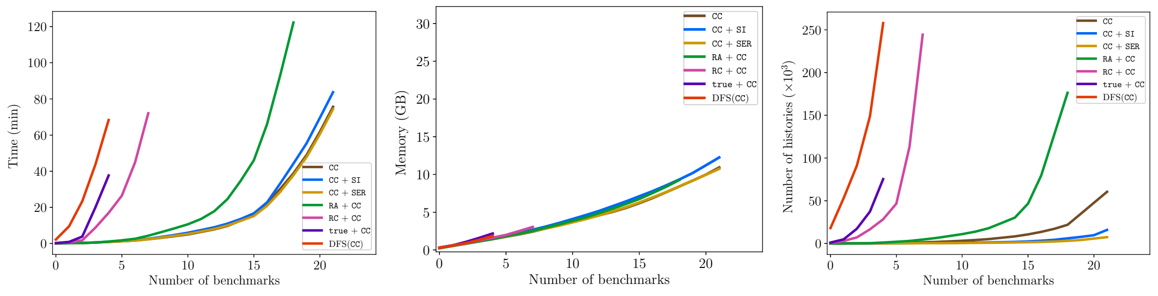
TPC-C [101] models an online shopping application with five types of transactions: reading the stock of a product, creating a new order, getting its status, paying it and delivering it.

SQL tables are modeled using a "set" global variable whose content is the set of ids (primary keys) of the rows present in the table, and a set of global variables, one variable for each row in the table (the name of the variable is the primary key of that row). SQL statements such as INSERT and DELETE statements are modeled as writes on that "set" variable while SQL statements with a WHERE clause (SELECT, JOIN, UPDATE) are compiled to a read of the table's set variable followed by reads or writes of variables that represent rows in the table (similarly to [30]).

3.7.3 Experimental Results

We designed three experiments where we compare the performance of a baseline model checking algorithm, EXPLORE-CE and EXPLORE-CE* for different (combinations of) isolation levels, and we explore the scalability of EXPLORE-CE when increasing the number of sessions and transactions per session, respectively. For each experiment we report running time, memory consumption, and the number of end states, i.e., histories of complete executions and in the case of EXPLORE-CE*, before applying the VALID filter. As the number of end states for a program on a certain isolation level increases, the running time of our algorithms naturally increases as well.

The first experiment compares the performance of our algorithms for different combinations of isolation levels and a baseline model checking algorithm that performs no partial order reduction. We consider as benchmark five (independent) client programs[5] for each application described above (25 in total), each program with 3 sessions and 3 transactions per session. Running time, memory consumption, and number of end states are reported in Fig. 3.15 as cactus plots [35].

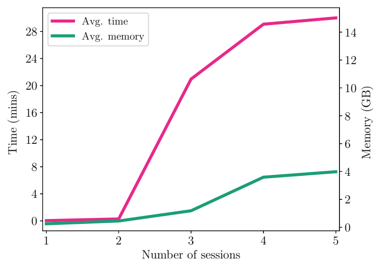


(a) Running time. (b) Memory consumption. (c) End states.

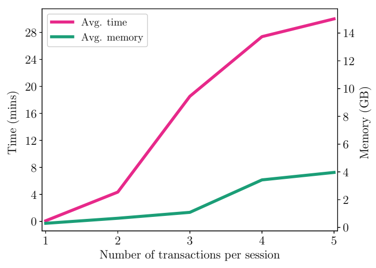
Figure 3.15: Cactus plots comparing different algorithms in terms of time, memory, and end states. For readability, we use TCC to denote EXPLORE-CE under TCC[6], $I_1 + I_2$ stands for EXPLORE-CE*(I_1, I_2), and true is the trivial isolation level where every history is consistent. Differences between TCC, TCC + SI and TCC + SER are very small and their graphics overlap. Moreover, DFS(TCC) denotes a standard DFS traversal of the semantics defined in Section 3.2.2. These plots exclude benchmarks that timeout (30 mins): 3 benchmarks for TCC, $\langle$SI, TCC$\rangle$ and $\langle$SER, TCC$\rangle$ and 6, 17, 20 and 20 benchmarks timeout for $\langle$RA, TCC$\rangle$, $\langle$RC, TCC$\rangle$, $\langle$true, TCC$\rangle$ and DFS(TCC) respectively.

To justify the benefits of partial order reduction, we implement a baseline model checking algorithm DFS(TCC) that performs a standard DFS traversal of the execution tree w.r.t. the formal semantics defined in Section 3.2.2 for TCC (for fairness, we restrict interleavings so at most one transaction is pending at a time). This baseline algorithm may explore the same history multiple times since it includes no partial order reduction mechanism. In terms of time, DFS(TCC) behaves poorly: it timeouts for 20 out of the 25 programs and it is less efficient even when it terminates. We consider a timeout of 30 mins. In comparison the strongly optimal algorithm EXPLORE-CE(TCC) (under TCC) finishes in $3'26''$ seconds in average (counting timeouts). DFS(TCC) is similiar to EXPLORE-CE(TCC) in terms of memory consumption. The memory consumption of DFS(TCC) is 381MB in average, compared to

[5]For an application that defines a number of transactions, a client program consists of a number of sessions, each session containing a sequence of transactions defined by the application.



(a) Increasing sessions.



(b) Increasing transactions per session.

Figure 3.16: Evaluating the scalability of EXPLORE-CE(TCC) for TPC-C and Wikipedia client programs when increasing their size. These plots include benchmarks that timeout (30 mins): 4, 9 and 10 for 3, 4 and 5 sessions respectively in Figure 3.16a, and 5, 8 and 10 for 3, 4 and 5 transactions per sessions respectively in Figure 3.16b.

508MB for EXPLORE-CE(TCC) (JPF forces a minimum consumption of 256MB).

To show the benefits of *strong* optimality, we compare EXPLORE-CE(TCC) which is strongly optimal with “plain” optimal algorithms EXPLORE-CE*(ι_0, TCC) for different levels ι_0. As shown in Figure 3.15a, EXPLORE-CE(TCC) is more efficient time-wise than every “plain” optimal algorithm, and the difference in performance grows as ι_0 becomes weaker. In the limit, when ι_0 is the trivial isolation level true where every history is consistent, EXPLORE-CE*(true, TCC) timeouts for 20 out of the 25 programs. The average speedup (average of individual speedups) of EXPLORE-CE(TCC) w.r.t. EXPLORE-CE*(RA, TCC), EXPLORE-CE*(RC, TCC) and EXPLORE-CE*(true, TCC) is 3, 18 and 15 respectively (we exclude timeout cases when computing speedups). All algorithms consume around 500MB of memory in average.

For the SI and SER isolation levels that admit no strongly optimal EXPLORE algorithm, we observe that the overhead of EXPLORE-CE*(TCC, SI) or EXPLORE-CE*(TCC, SER) relative to EXPLORE-CE(TCC) is negligible (the corresponding lines in Figure 3.15 are essentially overlapping). This is due to the fact that the consistency checking algorithms of [29] are polynomial time when the number of sessions is fixed, which makes them fast at least on histories with few sessions.

In our second experiment, we investigate the scalability of EXPLORE-CE when increasing the number of sessions. For each $i \in [1, 5]$, we consider 5 (independent) client programs for TPC-C and 5 for Wikipedia(10 in total) with ι sessions, each session containing 3 transactions. We start with 10 programs with 5 sessions, and remove sessions one by one to obtain programs with fewer sessions. We take TCC as isolation level. The plot in Figure 3.16a shows average running time and memory consumption for each number $i \in [1, 5]$ of sessions. As expected, increasing the number of sessions is a bottleneck running time wise because the number of histories increases significantly. However, memory consumption does not grow with the same trend, cf. the polynomial space bound.

Finally, we evaluate the scalability of EXPLORE-CE(TCC) when increasing the number of transactions per session. We consider 5 (independent) TPC-C client programs and 5 (inde-

[6]In the legend, CC is to be interpreted as TCC.

pendent) Wikipedia programs with 3 sessions and ι transactions per session, for each $i \in [1, 5]$. Figure 3.16b shows average running time and memory consumption for each number $i \in [1, 5]$ of transactions per session. Increasing the number of transactions per session is a bottleneck for the same reasons.

3.8 Related Work

Checking Correctness of Database-Backed Applications. One line of work is concerned with the logical formalization of isolation levels [105, 10, 27, 45, 29]. Our work relies on the axiomatic definitions of isolation levels introduced by [29], which have also investigated the problem of checking whether a given history satisfies a certain isolation level. Our SMC algorithms rely on these algorithms to check consistency of a history with a given isolation level.

Another line of work focuses on the problem of finding "anomalies": behaviors that are not possible under serializability. This is typically done via a static analysis of the application code that builds a static dependency graph that over-approximates the data dependencies in all possible executions of the application [44, 28, 53, 68, 104, 56]. Anomalies with respect to a given isolation level then correspond to a particular class of cycles in this graph. Static dependency graphs turn out to be highly imprecise in representing feasible executions, leading to false positives. Another source of false positives is that an anomaly might not be a bug because the application may already be designed to handle the non-serializable behavior [39, 56]. Recent work has tried to address these issues by using more precise logical encodings of the application [38, 39], or by using user-guided heuristics [56]. Another approach consists of modeling the application logic and the isolation level in first-order logic and relying on SMT solvers to search for anomalies [69, 86, 89], or defining specialized reductions to assertion checking [25, 24]. Our approach, based on SMC, does not generate false positives because we systematically enumerate only valid executions of a program which allows to check for user-defined assertions.

Several works have looked at the problem of reasoning about the correctness of applications executing under weak isolation and introducing additional synchronization when necessary [23, 64, 87, 77]. These are based on static analysis or logical proof arguments. The issue of repairing applications is orthogonal to our work.

MonkeyDB [30] is a mock storage system for testing storage-backed applications. While being able to scale to larger code, it has the inherent incompleteness of testing. As opposed to MonkeyDB, our algorithms perform a systematic and complete exploration of executions and can establish correctness at least in some bounded context, and they avoid redundancy, enumerating equivalent executions multiple times. Such guarantees are beyond the scope of MonkeyDB.

Dynamic Partial Order Reduction. [6] introduced the concept of *source sets* which provided the first strongly optimal DPOR algorithm for Mazurkiewicz trace equivalence. Other works study DPOR techniques for coarser equivalence relations, e.g., [8, 11, 17, 46, 47]. In all cases, the space complexity is exponential when strong optimality is ensured.

Other works focus on extending DPOR to weak memory models either by targeting a specific memory model [4, 5, 7, 88] or by being parametric with respect to an axiomaticallydefined memory model [72, 71, 73]. Some of these works can deal with the coarser reads-from equivalence, e.g., [7, 72, 71, 73]. Our algorithms build on the work of Kokologiannakis et al. [73] which for the first time, proposes a DPOR algorithm which is both strongly optimal and polynomial space. The definitions of database isolation levels are quite different with respect to weak memory models, which makes these previous works not extensible in a direct manner. These definitions include a semantics for *transactions* which are collections of reads and writes, and this poses new difficult challenges. For instance, reasoning about the completeness and the (strong) optimality of existing DPOR algorithms for shared-memory is agnostic to the scheduler (NEXT function) while the strong optimality of our EXPLORE-CE algorithm relies on the scheduler keeping at most one transaction pending at a time. In addition, unlike TruSt, EXPLORE-CE ensures that no swapped events can be swapped again and that the history order $<$ is an extension of $\mathsf{so} \cup \mathsf{wr}$. This makes our completeness and optimality proofs radically different. Moreover, even for transactional programs with one access per transaction, where $\mathtt{SER}$ and $\mathtt{SC}$ are equivalent, TruSt under $\mathtt{SC}$ and EXPLORE-CE$^*(\iota_0, \mathtt{SER})$ do not coincide, for any $\iota_0 \in \{\mathtt{RC}, \mathtt{RA}, \mathtt{TCC}\}$. In this case, TruSt enumerates only consistent histories w.r.t. $\mathtt{SC}$ at the cost of solving an NP-complete problem at each step while the EXPLORE-CE* step cost is polynomial time at the price of not being strongly-optimal. Furthermore, we identify isolation levels ($\mathtt{SI}$ and $\mathtt{SER}$) for which it is impossible to ensure both strong optimality and polynomial space bounds with a swapping-based algorithm, a type of question that has not been investigated in previous work.

3.9 Conclusions

We presented efficient SMC algorithms based on DPOR for transactional programs running under standard isolation levels. These algorithms are instances of a generic schema, called swapping-based algorithms, which is parametrized by an isolation level. Our algorithms are sound and complete, and polynomial space. Additionally, we identified a class of isolation levels, including $\mathtt{RC}$, $\mathtt{RA}$, and $\mathtt{TCC}$, for which our algorithms are strongly optimal, and we showed that swapping-based algorithms cannot be strongly optimal for stronger levels $\mathtt{SI}$ and $\mathtt{SER}$ (but just optimal). For the isolation levels we considered, there is an intriguing coincidence between the existence of a strongly optimal swapping-based algorithm and the complexity of checking if a given history is consistent with that level. Indeed, checking consistency is polynomial time for $\mathtt{RC}$, $\mathtt{RA}$, and $\mathtt{TCC}$, and NP-complete for $\mathtt{SI}$ and $\mathtt{SER}$. Investigating further the relationship between strong optimality and polynomial-time consistency checks is an interesting direction for future work.

4 On the Complexity of Testing SQL Transaction Isolation

4.1 Introduction

In this chapter, we focus on testing the isolation level implementations in databases, and more precisely, on the problem of checking whether a given execution adheres to the prescribed isolation level semantics.

As a first contribution, we introduce a formal axiomatic semantics for executions with SQL-like transactions with mixed isolation levels. We adapt the notion of history from Chapter 2 to this scenario. Dealing with SQL queries is more challenging than classic reads and writes of a *static* set of keys (as assumed in previous formalizations [45, 29]). SQL insert and delete queries change the set of locations at runtime and the set of locations returned by an SQL query depends on their values (the values are restricted to satisfy `WHERE` clauses).

We consider two classes of histories depending on the "completeness" of the write-read relation. To define a formal semantics of isolation levels, we need a complete write-read relation in the sense that for instance, an SQL select is associated with a write *for every* possible row (identified by its primary key) in the database, even if that row is *not* returned by the select because it does not satisfy the `WHERE` clause. Not returning a row is an observable effect that needs to be justified by the semantics. Such *full* histories can not be constructed by interacting with the database in a black-box manner (a desirable condition in testing) when only the outputs returned by queries can be observed. Therefore, we introduce the class of *client* histories where the write-read concerns only rows that are *returned* by a query. The consistency of a client history is defined as the existence of an extension of the write-read to a full history which satisfies the semantics. The semantics on full histories combines axioms from previous work [29] in a way that is directed by SQL queries that inspect the database and the isolation level of the transaction they belong to. This axiomatic semantics is validated by showing that it is satisfied by a standard operational semantics inspired by real implementations.

We study the complexity of checking if a full or client history is consistent, it satisfies the prescribed isolation levels. This problem is more complex for client histories, which record less dependencies and need to be extended to full ones.

For full histories, we show that the complexity of consistency checking matches previous results in the reads and writes model when all transactions have the same isolation level [29]: polynomial time for the so-called saturable isolation levels, and NP-complete for stronger levels like Snapshot Isolation or Serializability. The former is a new result that generalizes the work of [29] and exposes the key ideas for achieving polynomial-time complexity, while the latter is a consequence of the previous results.

We show that consistency checking becomes NP-complete for client histories even for saturable isolation levels. It remains NP-complete regardless of the expressiveness of `WHERE` clauses (for this stronger result we define another class of histories called *partial-observation*). The problem is NP-complete even if we bound the number of sessions. In general, transactions are organized in *sessions* [100], an abstraction of the sequence of transactions performed during the execution of an application (the counterpart of threads in shared memory). This case is interesting because it is polynomial-time in the read/write model [29].

As a counterpart to these negative results, we introduce an algorithm for checking consistency of client histories which is exponential-time in the worst case, but polynomial time in relevant cases. Given a client history as input, this algorithm combines an enumeration of extensions towards a full history with a search for a total commit order that satisfies the required axioms. The commit order represents the order in which transactions are committed in the database and it is an essential artifact for defining isolation levels. For efficiency, the algorithm uses a non-trivial enumeration of extensions that are *not* necessarily full but contain enough information to validate consistency. The search for a commit order is a non-trivial generalization of an algorithm by Biswas et al. [29] which concerned only serializability. This generalization applies to all practical isolation levels and combinations thereof. We evaluate an implementation of this algorithm on histories generated by PostgreSQL with a number of applications from BenchBase [51], e.g., the TPC-C model of a store and a model of Twitter. This evaluation shows that the algorithm is quite efficient in practice and scales well to typical workloads used in testing databases.

The rest of the chapter is structured as follows:

§ 4.2 presents the notions of full and client histories.

§ 4.3 introduces the SQL axiomatic semantics with mixed isolation levels and validate them with a standard operational semantics.

§ 4.4 lists multiple results on the complexity of checking consistency on full and client histories.

§ 4.5 presents an algorithm for effectively checking consistency of client histories that is exponential-time in worst-case scenarios but polynomial-time on relevant cases such as bounded conflict-free client histories.

§ 4.6 reports on an implementation and evaluation of our algorithm.

§ 4.7 concludes with a discussion of related work.

4.2 Transactional Programs with SQL-like Operations

In this section we describe the concrete program syntax employed during the rest of the chapter.

4.2.1 Program Syntax

Figure 4.1 lists the definition of a simple programming language that we use to represent applications running on top of a SQL-like database using mixed isolation levels. We model the database as a set of rows from an unbounded domain Rows. Each row is associated to a unique (primary) key from a domain Keys, given by the function $\mathsf{key} : \mathsf{Rows} \to \mathsf{Keys}$. Compared to Chapter 2, Keys and Rows are an alias of Objs and Vals respectively. In the following, we call keys to objects and rows to values.

We consider client programs accessing the database from a number of parallel sessions, each session being a sequence of transactions defined by the following grammar:

$$\iota \in \mathsf{Iso} \quad a \in \mathsf{LVars} \quad \mathsf{R} \in 2^{\mathsf{Rows}} \quad \mathsf{p} \in \mathsf{Rows} \to \{0,1\} \quad \mathsf{U} \in \mathsf{Keys} \to \mathsf{Rows}$$

$$\begin{aligned}
\mathsf{Transaction} &::= \mathtt{begin}(\iota); \mathsf{Body}; \mathtt{commit} \\
\mathsf{Body} &::= \mathsf{Instr} \ \mid \ \mathsf{Instr}; \mathsf{Body} \\
\mathsf{Instr} &::= \mathsf{InstrDB} \ \mid \ a := \mathsf{LExpr} \mid \ \mathtt{if}(\mathsf{LCond})\{\mathsf{Instr}\} \\
\mathsf{InstrDB} &::= a := \mathtt{SELECT}(\mathsf{p}) \ \mid \ \mathtt{INSERT}(\mathsf{R}) \ \mid \ \mathtt{DELETE}(\mathsf{p}) \ \mid \ \mathtt{UPDATE}(\mathsf{p}, \mathsf{U}) \ \mid \mathtt{abort}
\end{aligned}$$

Figure 4.1: Program syntax of a key-value store using SQL-like semantics.

For handling mixed isolation levels, the `begin` instruction defines an isolation level ι for the current transaction (see Section 2.4 for the description of the isolation levels). The body contains standard SQL-like statements for accessing the database and standard assignments and conditionals for local computation. As in Chapter 2, local computation uses (transaction-)local variables from a set LVars. We use a, b, ... to denote local variables. Expressions and Boolean conditions over local variables are denoted with LExpr and LCond, respectively.

Concerning database accesses (sometimes called queries), we consider a simplified but representative subset of SQL: $\mathtt{SELECT}(\mathsf{p})$ returns the set of rows satisfying the predicate p and the result is stored in a local variable a. $\mathtt{INSERT}(\mathsf{R})$ inserts the set of rows R or updates them in case they already exist (this corresponds to `INSERT ON CONFLICT DO UPDATE` in PostgreSQL), and $\mathtt{DELETE}(\mathsf{p})$ deletes all the rows that satisfy p. Then, $\mathtt{UPDATE}(\mathsf{p}, \mathsf{U})$ updates the rows satisfying p with values given by the map U, i.e., every row r in the database that satisfies p is replaced with $\mathsf{U}(\mathsf{key}(\mathsf{r}))$, and `abort` aborts the current transaction. The predicate p corresponds to a `WHERE` clause in standard SQL.

For an event e of type `SELECT`, `DELETE`, or `UPDATE`, we use $\mathtt{WHERE}(e)$ to denote the predicate p and for an `UPDATE` event e, we use $\mathtt{SET}(e)$ to denote the map U.

We call `read` events the `SELECT` events that read the database to return a set of rows, and the `DELETE` and `UPDATE` events that read the database checking satisfaction of some predicate p. Similarly, we call `write` events the `INSERT`, `DELETE` and `UPDATE` events that modify the database.

We consider transaction logs $(t, \iota_t, E, \mathsf{po}_t)$ as in Chapter 2, with additional *isolation level* identifiers, $\iota_t \in \mathsf{Iso}$, representing the isolation level declared by the unique `begin` event in E.

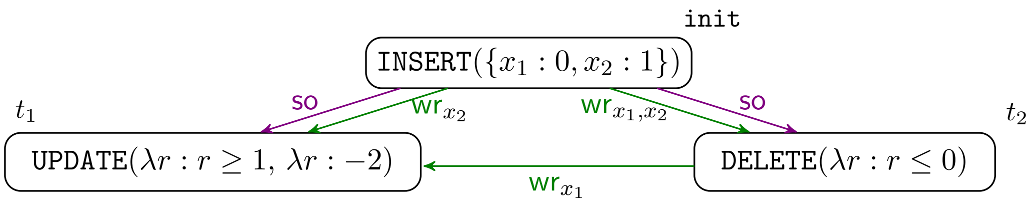

init
INSERT($\{x_1 : 0, x_2 : 1\}$)
t_1
UPDATE($\lambda r : r \geq 1,\ \lambda r : -2$)
t_2
DELETE($\lambda r : r \leq 0$)
so
so
wr_{x_2}
wr_{x_1,x_2}
wr_{x_1}

Figure 4.2: An example of a history (isolation levels omitted for legibility). Arrows represent so and wr relations. Transaction `init` defines the initial state: row 0 with key x_1 and row 1 with key x_2. Transaction t_2 reads x_1 and x_2 from `init` and deletes row with key x_1 (the only row satisfying predicate $\lambda r : r \leq 0$ corresponds to key x_1). Transaction t_1 reads x_1 from t_2 and x_2 from `init`, and updates only row with key x_2 as this is the only row satisfying predicate $\lambda r : r \geq 1$.

4.2.2 Histories

We present the specificities of histories using SQL-like semantics. The main difference with respect to the read-write semantics (Chapter 3) appears on the value function, as now $\texttt{value}_{\mathsf{wr}}(w, x)$ returns the row with key x written by the `write` event w. If w is an `INSERT`, it returns the inserted row with key x. If w is an `UPDATE`(p, U) event, it returns the value of U on key x if w reads a value for key x that satisfies predicate p. If w is a `DELETE`(p), it returns the special value $\dagger_x$ if w reads a value for key x that satisfies p. This special value indicates that the database does *not* contain a row with key x. We assume that the initial transaction is the only transaction that may insert as value $\dagger_x$ (indicating that initially, no row with key x is present). In case no condition is satisfied, $\texttt{value}_{\mathsf{wr}}(w, x)$ returns an undefined value $\perp$. We assume that the special values $\dagger_x$ or $\perp$ do not satisfy any predicate. Formally, the value function is described as follows:

$$\texttt{value}_{\mathsf{wr}}(w, x) = \begin{cases} \mathsf{r} & \text{if } w = \texttt{INSERT}(\mathsf{R}) \wedge \mathsf{r} \in \mathsf{R} \wedge \mathsf{key}(\mathsf{r}) = x \\ \dagger_x & \text{if } w = \texttt{DELETE}(\mathsf{p}) \wedge \mathsf{wr}_x^{-1}(w) \downarrow \wedge\ \mathsf{p}(\texttt{value}_{\mathsf{wr}}(\mathsf{wr}_x^{-1}(w), x)) = 1 \\ \mathsf{U}(x) & \text{if } w = \texttt{UPDATE}(\mathsf{p},\ \mathsf{U}) \wedge \mathsf{wr}_x^{-1}(w) \downarrow \wedge\ \mathsf{p}(\texttt{value}_{\mathsf{wr}}(\mathsf{wr}_x^{-1}(w), x)) = 1 \\ \perp & \text{otherwise} \end{cases}$$

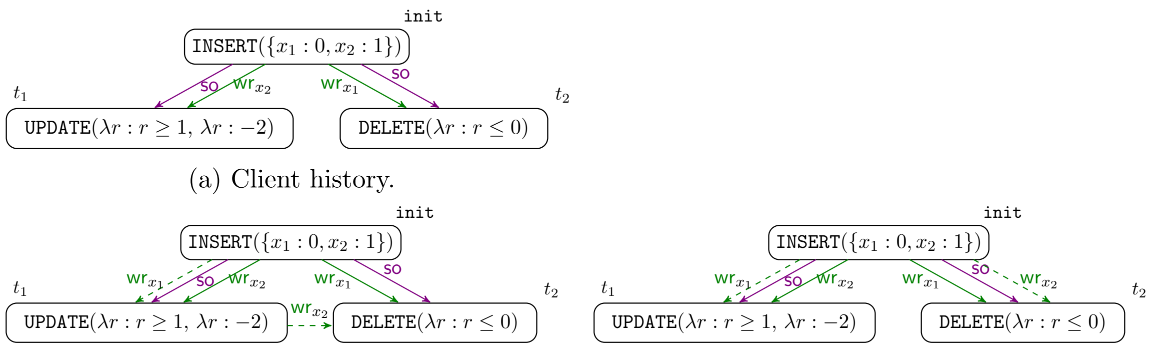
Note that the recursion in the definition of $\texttt{value}_{\mathsf{wr}}(w, x)$ terminates because wr is an acyclic relation.

Figure 4.2 shows an example of a history. For the `UPDATE` event w in t_1, $\texttt{value}_{\mathsf{wr}}(w, x_1) = \perp$ because this event reads x_1 from the `DELETE` event in t_2; while $\texttt{value}_{\mathsf{wr}}(w, x_2) = -2$ as it reads x_2 from the `INSERT` event in `init`.

4.2.3 Classes of Histories

We define two classes of histories: (1) *full* histories which are required to define the semantics of isolation levels and (2) *client* histories which model what is observable from interacting with a database as a black-box.

Full histories model the fact that every read query “inspects” an entire snapshot of the database in order to for instance, select rows satisfying some predicate. Roughly, full histories contain a write-read dependency for every read and key.


init
INSERT($\{x_1 : 0, x_2 : 1\}$)
t_1
t_2
so
wr_{x_2}
wr_{x_1}
UPDATE($\lambda r : r \geq 1,\ \lambda r : -2$)
DELETE($\lambda r : r \leq 0$)

(a) Client history.

(b) t_2 observes $x_2 = -2$.

(c) t_2 observes $x_2 = 1$.

Figure 4.3: Examples of a client history h and two possible extensions. The dashed edge belongs only to the extensions. The first extension is not a witness of h as t_1 writes -2 on x_2 and $\texttt{WHERE}(t_2)(-2) = 1$.

Definition 4.2.1. *A* full history $(T, \mathsf{so}, \mathsf{wr})$ *is a history where* $\mathsf{wr}_x^{-1}(r)$ *is defined for all* x *and* r*, unless* r *reads locally.*

Client histories record less write-read dependencies compared to full histories, which is formalized by the *extends* relation.

Definition 4.2.2. *A history* $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ extends *another history* $h = (T, \mathsf{so}, \mathsf{wr})$ *if* $\mathsf{wr}_x \subseteq \overline{\mathsf{wr}}_x$*. We denote it by* $h \subseteq \overline{h}$*.*

Definition 4.2.3. *A* client history $h = (T, \mathsf{so}, \mathsf{wr})$ *is a history s.t. there is a full history* $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ *with* $h \subseteq \overline{h}$*, and s.t for every* x*, if* $(w, r) \in \overline{\mathsf{wr}}_x \setminus \mathsf{wr}_x$*, then* $\texttt{WHERE}(r)(\texttt{value}_{\overline{\mathsf{wr}}}(w, x)) = 0$*. The history* $\overline{h}$ *is called a* witness *of* h*.*

Compared to a witness full history, a client history may omit write-read dependencies if the written values do *not* satisfy the predicate of the read query. These values would not be observable when interacting with the database as a black-box. This includes the case when the write is a DELETE (recall that the special value $\dagger_x$ indicating deleted rows falsifies every predicate by convention). Figure 4.2 shows a full history as every query reads both x_1 and x_2. Figure 4.3a shows a client history: transactions t_1, t_2 does not read x_2 and x_1 resp. Figure 4.3b is an extension but not a witness while Figure 4.3c is indeed a witness of it.

4.3 Axiomatic Semantics with Different Isolation Levels

We define an axiomatic semantics on histories where transactions can be assigned different isolation levels, which builds on Chapter 2.

4.3.1 Isolation Configurations

The *isolation configuration* of a history is a mapping $\mathsf{iso}(h) : T \to \mathsf{Iso}$ associating to each transaction its isolation level identifier. Whenever every transaction in a history has the same isolation level ι, the isolation configuration of that history is denoted simply by ι.

In general, for two isolation configurations I_1 and I_2, I_1 is *stronger than* I_2 when for every transaction t, $I_1(t)$ is stronger than $I_2(t)$ (i.e., whenever $I_1(t)$ holds in an execution ξ, $I_2(t)$ also holds in ξ). The *weaker than* relationship is defined similarly.

Given a full history h with isolation configuration $\mathsf{iso}(h)$, h is called *consistent* (with respect to its isolation configuration) when there exists an execution ξ of h such that for all transactions t in ξ, the axioms in $\mathsf{iso}(h)(t)$ are satisfied in ξ. Unlike in Chapter 3, the constraints imposed by axiom a on event r (i.e. the conditions on co for ensuring that $a(r)$ holds in ξ) only apply whenever the axiom a is an axiom of the isolation level declared by $\mathsf{tr}(r)$.

For example, let h be the full history in Figure 4.3c. If both t_1, t_2's isolation are $\mathtt{SER}$, then h is *not* consistent, i.e., every execution $\xi = (h, \mathsf{co})$ violates the corresponding axioms. Assume for instance, that $(t_1, t_2) \in \mathsf{co}$. Then, by axiom $\mathtt{SER}$, as $(\mathtt{init}, t_2) \in \mathsf{wr}_{x_1}$ and t_1 writes x_1, we get that $(t_1, \mathtt{init}) \in \mathsf{co}$, which is impossible as $(\mathtt{init}, t_1) \in \mathsf{so} \subseteq \mathsf{co}$. However, if the isolation configuration is weaker (for example $\mathsf{iso}(h)(t_2) = \mathtt{RC}$), then the history is consistent using $\mathtt{init} <_{\mathsf{co}} t_1 <_{\mathsf{co}} t_2$ as commit order.

Definition 4.3.1. *A full history $h = (T, \mathsf{so}, \mathsf{wr})$ with isolation configuration* $\mathsf{iso}(h)$ *is* consistent *iff there is an execution ξ of h s.t. $\bigwedge_{t \in T, r \in \mathsf{reads}(t), a \in \mathsf{iso}(h)(t)} a(r)$ holds in ξ; ξ is called a* consistent *execution of h.*

The notion of consistency on full histories is extended to client histories.

Definition 4.3.2. *A client history $h = (T, \mathsf{so}, \mathsf{wr})$ with isolation configuration* $\mathsf{iso}(h)$ *is* consistent *iff there is a full history $\overline{h}$ with the same isolation configuration which is a witness of h and consistent; $\overline{h}$ is called a* consistent *witness of h.*

In general, the witness of a client history may not be consistent. In particular, there may exist several witnesses but no consistent witness.

4.3.2 Validation of the semantics

To justify the axiomatic semantics defined above, we define an operational semantics inspired by real implementations and prove that every run of a program can be translated into a consistent history. Every instruction is associated with an increasing timestamp and it reads from a snapshot of the database defined according to the isolation level of the enclosing transaction. At the end of the transaction we evaluate if the transaction can be committed or not. We assume that a transaction can abort only if explicitly stated in the program. We model an optimistic approach where if a transaction cannot commit, the run blocks (modelling unexpected aborts). We focus on three of the most used isolation levels: $\mathtt{SER}, \mathtt{SI}, \mathtt{RC}$. Other isolation levels can be handled in a similar manner. For each run ρ we extract a full history $\mathsf{history}(\rho)$. We show by induction that $\mathsf{history}(\rho)$ is consistent at every step.

Formally, the operational semantics is defined as a transition relation $\Rightarrow$ between *configurations*. A configuration is a tuple containing the following:

- history h recording the instructions executed in the past, BEGIN

$$\frac{\begin{array}{c} t \text{ fresh} \quad e \text{ fresh} \quad \mathsf{P}(j) = \mathtt{begin}(\iota); \mathsf{Body}; \mathtt{commit}; \mathsf{P} \quad \vec{\mathsf{B}}(j) = \epsilon \\ \tau = 1 + \max\{\vec{\mathsf{T}}(e') \mid e' \in \mathsf{events}(h)\} \quad \vec{\mathsf{T}}' = \vec{\mathsf{T}}[e \to \tau] \\ \delta = \mathsf{snapshot}_\iota(h, \vec{\mathsf{S}}, \vec{\mathsf{T}}', e, \mathtt{begin}) \quad h' = h \oplus_j (t, \iota, \{(e, \mathtt{begin})\}, \emptyset) \end{array}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \vec{\mathsf{I}}, \vec{\mathsf{T}}, \vec{\mathsf{S}}, \mathsf{P} \Rightarrow h', \vec{\gamma}[j \mapsto \emptyset], \vec{\mathsf{B}}[j \mapsto \mathsf{Body}; \mathtt{commit}], \vec{\mathsf{I}}[t \mapsto \iota], \vec{\mathsf{T}}', \vec{\mathsf{S}}[e \mapsto \delta], \mathsf{P}[j \mapsto \mathsf{S}]}$$

IF-TRUE

$$\frac{\psi(\vec{a})[\vec{\gamma}(j)(a)/a : a \in \vec{a}] \quad \vec{\mathsf{B}}(j) = \mathtt{if}(\psi(\vec{a}))\{\mathsf{Instr}\}; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \vec{\mathsf{I}}, \vec{\mathsf{T}}, \vec{\mathsf{S}}, \mathsf{P} \Rightarrow h, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{Instr}; \mathsf{B}], \vec{\mathsf{I}}, \vec{\mathsf{T}}, \vec{\mathsf{S}}, \mathsf{P}}$$

IF-FALSE

$$\frac{\neg\psi(\vec{a})[\vec{\gamma}(j)(a)/a : a \in \vec{a}] \quad \vec{\mathsf{B}}(j) = \mathtt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \vec{\mathsf{I}}, \vec{\mathsf{T}}, \vec{\mathsf{S}}, \mathsf{P} \Rightarrow h, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \vec{\mathsf{I}}, \vec{\mathsf{T}}, \vec{\mathsf{S}}, \mathsf{P}}$$

LOCAL

$$\frac{v = e[\vec{\gamma}(j)(a')/a' : a' \in \vec{a'}] \quad \vec{\mathsf{B}}(j) = a := e(\vec{a'}); \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \vec{\mathsf{I}}, \vec{\mathsf{T}}, \vec{\mathsf{S}}, \mathsf{P} \Rightarrow h, \vec{\gamma}[j, a \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \vec{\mathsf{I}}, \vec{\mathsf{T}}, \vec{\mathsf{S}}, \mathsf{P}}$$

COMMIT

$$\frac{\begin{array}{c} \vec{\mathsf{B}}(j) = \mathtt{commit} \quad e, t, \iota, \tau = \mathsf{next}(h, j, \vec{\mathsf{T}}) \\ \vec{\mathsf{T}}' = \vec{\mathsf{T}}[e \to \tau] \qquad \delta = \mathsf{snapshot}_\iota(h, \vec{\mathsf{S}}, \vec{\mathsf{T}}', e, \mathtt{commit}) \quad \mathsf{validate}_\iota(h, \vec{\mathsf{T}}', t) \end{array}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \vec{\mathsf{I}}, \vec{\mathsf{T}}, \mathsf{P} \Rightarrow h \oplus_j (e, \mathtt{commit}), \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \epsilon], \vec{\mathsf{I}}, \vec{\mathsf{T}}', \vec{\mathsf{S}}[e \mapsto \delta], \mathsf{P}}$$

ABORT

$$\frac{\begin{array}{c} \vec{\mathsf{B}}(j) = \mathsf{abort}; B \quad e, t, \iota, \tau = \mathsf{next}(h, j, \vec{\mathsf{T}}) \\ \vec{\mathsf{T}}' = \vec{\mathsf{T}}[e \to \tau] \qquad \delta = \mathsf{snapshot}_\iota(h, \vec{\mathsf{S}}, \vec{\mathsf{T}}', e, \mathtt{abort}) \quad \mathsf{validate}_\iota(h, \vec{\mathsf{T}}', t) \end{array}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \vec{\mathsf{I}}, \vec{\mathsf{T}}, \vec{\mathsf{S}}, \mathsf{P} \Rightarrow h \oplus_j (e, \mathtt{abort}), \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \epsilon], \vec{\mathsf{I}}, \vec{\mathsf{T}}', \vec{\mathsf{S}}[e \mapsto \delta], \mathsf{P}}$$

Figure 4.4: An operational semantics for transactional programs. Above, next, computes the next event, transaction, isolation level and timestamp employed respectively, while $\mathsf{snapshot}_\iota$ and readFrom denotes the snapshot visible to an instruction and the writes it reads from, respectively. The $\mathsf{validate}_\iota$ predicate checks if a transaction can be committed. They are defined in Figure 4.6.

.

- a valuation map $\vec{\gamma}$ that records local variable values in the current transaction of each session ($\vec{\gamma}$ associates identifiers of sessions that have live transactions with valuations of local variables),
- a map $\vec{\mathsf{B}}$ that stores the code of each live transaction (associating session identifiers with code),
- a map $\vec{\mathsf{I}}$ that tracks the isolation level of each executed transaction,

$$\textsc{Insert}\quad \frac{\begin{array}{c}\vec{\mathsf{B}}(j) = \mathtt{INSERT}(\mathsf{R});\mathsf{B}\quad e,t,\iota,\tau = \mathsf{next}(h,j,\vec{\mathsf{T}})\\ \vec{\mathsf{T}}' = \vec{\mathsf{T}}[e \to \tau]\qquad \delta = \mathsf{snapshot}_\iota(h,\vec{\mathsf{S}},\vec{\mathsf{T}}',e,\mathtt{INSERT})\quad h' = h \oplus_j (e,\mathtt{INSERT}(\mathsf{R}))\end{array}}{h,\vec{\gamma},\vec{\mathsf{B}},\vec{\mathsf{I}},\vec{\mathsf{T}},\vec{\mathsf{S}},\mathsf{P} \Rightarrow h',\vec{\gamma},\vec{\mathsf{B}}[j \mapsto \mathsf{B}],\vec{\mathsf{I}},\vec{\mathsf{T}}',\vec{\mathsf{S}}[e \mapsto \delta],\mathsf{P}}$$

$$\textsc{Select}\quad \frac{\begin{array}{c}\vec{\mathsf{B}}(j) = a := \mathtt{SELECT}(\mathsf{p});\mathsf{B}\quad e,t,\iota,\tau = \mathsf{next}(h,j,\vec{\mathsf{T}})\\ \vec{\mathsf{T}}' = \vec{\mathsf{T}}[e \to \tau]\qquad \delta = \mathsf{snapshot}_\iota(h,\vec{\mathsf{S}},\vec{\mathsf{T}}',e,\mathtt{SELECT})\quad \vec{\mathsf{w}} = \mathsf{readFrom}(h,\vec{\mathsf{T}},t,\delta)\\ h' = (h \oplus_j (e,\mathtt{SELECT}(\mathsf{p}))) \bigoplus_{x \in \mathsf{Keys}, \vec{\mathsf{w}}[x] \neq \bot} \mathsf{wr}(\vec{\mathsf{w}}[x],e)\end{array}}{h,\vec{\gamma},\vec{\mathsf{B}},\vec{\mathsf{I}},\vec{\mathsf{T}},\vec{\mathsf{S}},\mathsf{P} \Rightarrow h',\vec{\gamma}[(j,a) \mapsto \{\mathsf{r} \in \delta : \mathsf{p}(\mathsf{r})\}],\vec{\mathsf{B}}[j \mapsto \mathsf{B}],\vec{\mathsf{I}},\vec{\mathsf{T}}',\vec{\mathsf{S}}[e \mapsto \delta],\mathsf{P}}$$

$$\textsc{Update}\quad \frac{\begin{array}{c}\vec{\mathsf{B}}(j) = \mathtt{UPDATE}(\mathsf{p},\ \mathsf{U});\mathsf{B}\quad e,t,\iota,\tau = \mathsf{next}(h,j,\vec{\mathsf{T}})\\ \vec{\mathsf{T}}' = \vec{\mathsf{T}}[e \to \tau]\qquad \delta = \mathsf{snapshot}_\iota(h,\vec{\mathsf{S}},\vec{\mathsf{T}}',e,\mathtt{UPDATE})\quad \vec{\mathsf{w}} = \mathsf{readFrom}(h,\vec{\mathsf{T}},t,\delta)\\ h' = (h \oplus_j (e,\mathtt{UPDATE}(\mathsf{p},\ \mathsf{U}))) \bigoplus_{x \in \mathsf{Keys}, \vec{\mathsf{w}}[x] \neq \bot} \mathsf{wr}(\vec{\mathsf{w}}[x],e)\end{array}}{h,\vec{\gamma},\vec{\mathsf{B}},\vec{\mathsf{I}},\vec{\mathsf{T}},\vec{\mathsf{S}},\mathsf{P} \Rightarrow h',\vec{\gamma},\vec{\mathsf{B}}[j \mapsto \mathsf{B}],\vec{\mathsf{I}},\vec{\mathsf{T}}',\vec{\mathsf{S}}[e \mapsto \delta],\mathsf{P}}$$

$$\textsc{Delete}\quad \frac{\begin{array}{c}\vec{\mathsf{B}}(j) = \mathtt{DELETE}(\mathsf{p});\mathsf{B}\quad e,t,\iota,\tau = \mathsf{next}(h,j,\vec{\mathsf{T}})\\ \vec{\mathsf{T}}' = \vec{\mathsf{T}}[e \to \tau]\qquad \delta = \mathsf{snapshot}_\iota(h,\vec{\mathsf{S}},\vec{\mathsf{T}}',e,\mathtt{DELETE})\quad \vec{\mathsf{w}} = \mathsf{readFrom}(h,\vec{\mathsf{T}},t,\delta)\\ h' = (h \oplus_j (e,\mathtt{DELETE}(\mathsf{p}))) \bigoplus_{x \in \mathsf{Keys}, \vec{\mathsf{w}}[x] \neq \bot} \mathsf{wr}(\vec{\mathsf{w}}[x],e)\end{array}}{h,\vec{\gamma},\vec{\mathsf{B}},\vec{\mathsf{I}},\vec{\mathsf{T}},\vec{\mathsf{S}},\mathsf{P} \Rightarrow h',\vec{\gamma},\vec{\mathsf{B}}[j \mapsto \mathsf{B}],\vec{\mathsf{I}},\vec{\mathsf{T}}',\vec{\mathsf{S}}[e \mapsto \delta],\mathsf{P}}$$

Figure 4.5: An operational semantics for transactional programs. Above, next, computes the next event, transaction, isolation level and timestamp employed respectively, while $\mathsf{snapshot}_\iota$ and readFrom denotes the snapshot visible to an instruction and the writes it reads from, respectively. The $\mathsf{validate}_\iota$ checks if a transaction can be committed. They are defined in Figure 4.6.

.

- a map $\vec{\mathsf{T}}$ that associates events in the history with unique timestamps,
- a map $\vec{\mathsf{S}}$ that associates events in the history with snapshots of the database,
- sessions/transactions P that remain to be executed from the original program.

For readability, we define a program as a partial function $\mathsf{P} : \mathsf{SessId} \rightharpoonup \mathsf{Sess}$ that associates session identifiers in SessId with sequences of transactions as defined in Section 4.2.1. Similarly, the session order so in a history is defined as a partial function $\mathsf{so} : \mathsf{SessId} \rightharpoonup \mathsf{Tlogs}^*$ that associates session identifiers with sequences of transaction logs. Two transaction logs are ordered by so if one occurs before the other in some sequence $\mathsf{so}(j)$ with $j \in \mathsf{SessId}$.

Before presenting the definition of $\Rightarrow_I$, we introduce some notation. Let h be a history that contains a representation of so as above. We use $h \oplus_j (t, \iota_t, E, \mathsf{po}_t)$ to denote a history where $(t, \iota_t, E, \mathsf{po}_t)$ is appended to $\mathsf{so}(j)$. Also, for an event e, $h \oplus_j e$ is the history obtained from h by adding e to the last transaction log in $\mathsf{so}(j)$ and as a last event in the program order of this log (i.e., if $\mathsf{so}(j) = \sigma; (t, \iota_t, E, \mathsf{po}_t)$, then the session order so' of $h \oplus_j e$ is defined by $\mathsf{so}'(k) = \mathsf{so}(k)$ for all $k \neq j$ and $\mathsf{so}(j) = \sigma; (t, \iota_t, E \cup e, \mathsf{po} \cup \{(e', e) : e' \in E\})$). Finally, for a history $h = (T, \mathsf{so}, \mathsf{wr})$, $h \oplus \mathsf{wr}(t, e)$ is the history obtained from h by adding (t, e) to the write-read relation.

Figures 4.4 and 4.5 list the rules defining $\Rightarrow_I$. We distinguish between local computation rules (IF-TRUE, IF-FALSE and LOCAL) and database-accesses rules (BEGIN, INSERT, SELECT, UPDATE, DELETE, COMMIT and ABORT); each associated to its homonymous instruction. Database-accesses get an increasing timestamp τ as well as an isolation-depending snapshot of the database using predicate $\mathsf{snapshot}_\iota$; updating adequately the timestamp and snapshot maps ($\vec{\mathsf{T}}$ and $\vec{\mathsf{S}}$ respectively). Timestamps are used for validating the writes of a transaction and blocking inconsistent runs as well as for defining the set of possible snapshots any event can get. We use predicate $\mathsf{readFrom}$ for determining the values read by an event. Those reads depend on both the event's snapshot as well as the timestamp of every previously executed event. Their formal definitions are described in Figure 4.6.

The BEGIN rule starts a new transaction, provided that there is no other live transaction ($\mathsf{B} = \epsilon$) in the same session. It adds an empty transaction log to the history and schedules the body of the transaction. IF-TRUE and IF-FALSE check the truth value of a Boolean condition of an `if` conditional. LOCAL handles the case where some local computation is required. INSERT, SELECT, UPDATE and DELETE handle the database accesses. INSERT add some rows R in the history. SELECT, UPDATE and DELETE read every key from a combination of its snapshot and the local writes defined by $\mathsf{readFrom}$ function. The predicate $\_ \ \mathsf{writes}\ \_$ implicitly uses the previous information stored in the history via the function $\mathsf{value}_{\mathsf{wr}}$. Finally COMMIT and ABORT validate that the run of the transaction correspond to the isolation level specification. These rules may block in case the validation is not satisfied as the predicate valuation does not change with the application of posterior rules.

An *initial* configuration for program P contains the program P along with a history $h = (\{t_0\}, \emptyset, \emptyset)$, where t_0 is a transaction log containing only writes that write the initial values of all keys and whose timestamp and snapshot is 0 ($\vec{\mathsf{S}}, \vec{\mathsf{T}} = [t_0 \mapsto 0]$), and it does not contain transaction code nor local keys ($\vec{\gamma}, \vec{\mathsf{B}} = \emptyset$). A *run* ρ of a program P is a sequence of configurations $c_0 c_1 \ldots c_n$ where c_0 is an initial configuration for P, and $c_m \Rightarrow c_{m+1}$, for every $0 \leq m < n$. We say that c_n is *reachable* from c_0. The history of such a run, $\mathsf{history}(\rho)$, is the history h_n in the last configuration c_n. A configuration is called *final* if it contains the empty program ($\mathsf{P} = \emptyset$). Let $\mathsf{hist}(\mathsf{P})$ denote the set of all histories of a run of P that ends in a final configuration.

The following theorem states the validation of our axiomatic semantics:

Theorem 4.3.3. *For every run ρ, $\mathsf{history}(\rho)$ is a consistent full history.*

The proof of Theorem 4.3.3 is split in two parts: Lemma 4.3.5 and Lemma 4.3.7. In

$$
\begin{array}{rl}
e, t, \iota, \tau = & \mathsf{next}(h, j, \vec{\mathsf{T}}) \text{ where} \\
& \quad e \text{ fresh }, \ t = \mathtt{last}(h, j), \ \iota = \mathsf{iso}(h)(t) \text{ and } \tau = 1 + \max\{\vec{\mathsf{T}}(e') \mid e' \in \mathsf{events}(h)\}
\end{array}
$$

$$
\mathsf{snapshot}_{\mathtt{SER}}(h, \vec{\mathsf{S}}, \vec{\mathsf{T}}', e, \xi) = \begin{cases} \max \left\{ \vec{\mathsf{T}}'(c_{t'}) \;\middle|\; \begin{array}{l} t' \in h \ \wedge \\ c_{t'} = \mathtt{commit}(t') \ \wedge \\ \vec{\mathsf{T}}'(c_{t'}) < \vec{\mathsf{T}}'(e) \end{array} \right\} & \text{if } \xi = \mathtt{begin} \\ \vec{\mathsf{S}}(\mathtt{begin}(\mathsf{tr}(e))) & \text{otherwise} \end{cases}
$$

$$
\mathsf{snapshot}_{\mathtt{SI}}(h, \vec{\mathsf{S}}, \vec{\mathsf{T}}', e, \xi) = \begin{cases} \mathsf{choice}\left(\left\{ \vec{\mathsf{T}}'(c_{t'}) \;\middle|\; \begin{array}{l} t' \in h \ \wedge \\ c_{t'} = \mathtt{commit}(t') \ \wedge \\ vec\mathsf{T}'(c_{t'}) < \vec{\mathsf{T}}'(e) \end{array} \right\} \right) & \text{if } \xi = \mathtt{begin} \\ \vec{\mathsf{S}}(\mathtt{begin}(\mathsf{tr}(e))) & \text{otherwise} \end{cases}
$$

$$
\mathsf{snapshot}_{\mathtt{RC}}(h, \vec{\mathsf{S}}, \vec{\mathsf{T}}', e, \xi) = \quad \mathsf{choice}\left(\left\{ \vec{\mathsf{T}}'(c_{t'}) \;\middle|\; \begin{array}{l} t' \in h \wedge \\ c_{t'} = \mathtt{commit}(t') \wedge \vec{\mathsf{T}}'(c_{t'}) < \vec{\mathsf{T}}'(e) \wedge \\ \forall e'. \left(\begin{array}{l} (e', e) \in \mathsf{po} \ \vee \\ (\mathsf{tr}(e'), \mathsf{tr}(e)) \in \mathsf{so} \end{array} \right) \\ \qquad \implies \vec{\mathsf{S}}(e') \leq \vec{\mathsf{T}}(c_{t'}) \end{array} \right\} \right)
$$

$$
\begin{array}{rl}
\mathsf{readFrom}(h, \vec{\mathsf{T}}, t, \delta) = & [x \mapsto (\mathtt{localWr}[x] \neq \bot)\,?\,\bot \ : \ w_x \text{ for each } x \in \mathsf{Keys}] \\
\text{where } \mathtt{localWr}[x] = & \max_{\mathsf{po}}\{e \mid \mathsf{tr}(e) = t \ \wedge \ e \ \mathsf{writes}\ x\} \cup \{\bot\} \\
\text{and } w_x \ \mathsf{writes}\ x \ \wedge & \vec{\mathsf{T}}(w_x) = \max \left\{ \vec{\mathsf{T}}(w') \;\middle|\; \begin{array}{c} w' \in \mathsf{events}(h) \ \wedge \ w' \ \mathsf{writes}\ x \ \wedge \\ \vec{\mathsf{T}}(\mathtt{commit}(\mathsf{tr}(w'))) \leq \delta \end{array} \right\}
\end{array}
$$

$$
\mathsf{validate}_{\mathtt{SER}}(h, \vec{\mathsf{T}}', t) = \left(\begin{array}{l} \nexists t' \in h, x \in \mathsf{Keys} \text{ s.t. } (t \ \mathsf{reads}\ x \vee t \ \mathsf{writes}\ x) \ \wedge \ t' \ \mathsf{writes}\ x \\ \quad \wedge \ \vec{\mathsf{T}}'(\mathtt{begin}(t)) < \vec{\mathsf{T}}'(\mathtt{commit}(t')) < \vec{\mathsf{T}}'(\mathtt{end}(t)) \end{array} \right)
$$

$$
\mathsf{validate}_{\mathtt{SI}}(h, \vec{\mathsf{T}}', t) = \left(\begin{array}{l} \nexists t' \in h, x \in \mathsf{Keys} \text{ s.t. } t \ \mathsf{writes}\ x \ \wedge \ t' \ \mathsf{writes}\ x \ \wedge \\ \quad \wedge \ \vec{\mathsf{T}}'(\mathtt{begin}(t)) < \vec{\mathsf{T}}'(\mathtt{commit}(t')) < \vec{\mathsf{T}}'(\mathtt{end}(t)) \end{array} \right)
$$

$$
\mathsf{validate}_{\mathtt{RC}}(h, \vec{\mathsf{T}}', t) = \quad \mathsf{true}
$$

Figure 4.6: Definition of auxiliary functions for the operational semantics. The function choice receives a set as input and returns one of its elements.

.

Lemma 4.3.5, we prove by induction that for any run ρ, $\mathsf{history}(\rho)$ is a full history; using the auxiliary Lemma 4.3.4 about pending transactions. We then define in Equation 4.1 a relation on transactions that plays the role of consistency witness for $\mathsf{history}(\rho)$. Then, we prove in Lemma 4.3.6 that such relation is a commit order for $\mathsf{history}(\rho)$ to conclude in Lemma 4.3.7 that $\mathsf{history}(\rho)$ is indeed consistent. In all cases, we do a case-by-case analysis depending on which rule is employed during the inductive step.

For the sake of simplifying our notation, we denote by $\mathsf{rule}(\rho, j, \rho')$ to the rule s.t. applied to run ρ on session j leads to configuration ρ'.

Lemma 4.3.4. *Let ρ be a run and* $\mathsf{history}(\rho) = (T, \mathsf{so}, \mathsf{wr})$ *be its history. Any pending transaction in T is* $(\mathsf{so} \cup \mathsf{wr})$*-maximal.*

Proof. We prove by induction on the length of a run ρ that any pending transaction is $(\mathsf{so}{\cup}\mathsf{wr})$-maximal; where $\mathsf{history}(\rho) = (T, \mathsf{so}, \mathsf{wr})$. The base case, where $\rho = \{c_0\}$ and c_0 is an initial configuration, is immediate by definition. Let us suppose that for every run of length at most n the property holds and let ρ' a run of length $n+1$. As ρ' is a sequence of configurations, there exist a reachable run ρ of length n, a session j and a rule r s.t. $r = \mathsf{rule}(\rho, j, \rho')$. Let us call $h = (T, \mathsf{so}, \mathsf{wr})$, $h' = (T', \mathsf{so}', \mathsf{wr}')$ and e to $\mathsf{history}(\rho)$, $\mathsf{history}(\rho')$ and the last event in po-order belonging to $\mathtt{last}(h, j)$ respectively. By induction hypothesis, any pending transaction in h is $(\mathsf{so} \cup \mathsf{wr})$-maximal. To conclude the inductive step, we show that for every possible rule r s.t. $r = \mathsf{rule}(\rho, j, \rho')$, the property also holds in h'.

- LOCAL, IF-FALSE, IF-TRUE, INSERT, COMMIT, ABORT: The result trivially holds as $\mathsf{wr}' = \mathsf{wr}$, $\mathsf{so}' = \mathsf{so}$ and $\mathsf{complete}(T') \subseteq \mathsf{complete}(T)$.

- BEGIN: We observe that in this case, $T = T \cup \{\mathtt{last}(h, j)\}$, $\mathsf{reads}(T') = \mathsf{reads}(T)$, $\mathsf{wr}' = \mathsf{wr}$ and $\mathsf{so}' = \mathsf{so} \cup \{(t', \mathtt{last}(h, j)) \mid \mathtt{ses}(t') = j\}$. Thus, $\mathtt{last}(h, j)$ is pending and $\mathsf{so}'{\cup}\mathsf{wr}'$-maximal. Moreover, as described in Figure 4.4, $\vec{\mathsf{B}}(j) = \epsilon$; so there is no other transaction in session j that is pending. Hence, as $T' \setminus \mathsf{complete}(T') = T\mathsf{complete}(T) \cup \{\mathtt{last}(h, j)\}$, by induction hypothesis, every pending transaction is $\mathsf{so}' \cup \mathsf{wr}'$-maximal.

- SELECT, UPDATE, DELETE: Figure 4.5 describes h' by the equation $h' = (h \oplus_j (e, \mathsf{rule}(\rho, j, s))) \bigoplus_{x \in \mathsf{Keys}, \vec{\mathsf{w}}[x] \neq \perp} \mathsf{wr}(\vec{\mathsf{w}}[x], e)$; where e is the new event executed and $\vec{\mathsf{w}}$ is defined following the descriptions in Figures 4.5 and 4.6. In this case, $T' = T$, $\mathsf{reads}(T') = \mathsf{reads}(T) \cup \{e\}$, $\mathsf{so}' = \mathsf{so}$, $\forall x \in \mathsf{Keys}$ s.t. $\vec{\mathsf{w}}[x] = \perp$, $\mathsf{wr}'_x = \mathsf{wr}_x$ and $\forall x \in \mathsf{Keys}$ s.t. $\vec{\mathsf{w}}[x] \neq \perp$, $\mathsf{wr}'_x = \mathsf{wr}_x \cup \{(\vec{\mathsf{w}}[x], e)\}$. Note that as described by Figure 4.6, in the latter case, when $\vec{\mathsf{w}}[x] \neq \perp$, $\mathsf{tr}(\vec{\mathsf{w}}[x]) \in \mathsf{complete}(T) = \mathsf{complete}(T')$. In conclusion, using the induction hypothesis, we also conclude that every pending transaction is $\mathsf{so}' \cup \mathsf{wr}'$-maximal.

□

Lemma 4.3.5. *For every run ρ,* $\mathsf{history}(\rho)$ *is a full history.*

Proof. We prove by induction on the length of a run ρ that $\mathsf{history}(\rho)$ is a full history; where the base case, $\rho = \{c_0\}$ and c_0 is an initial configuration, is trivial by definition. Let us suppose that for every run of length at most n the property holds and let ρ' a run of length $n+1$. As ρ' is a sequence of configurations, there exist a reachable run ρ of length n, a session j and a rule r s.t. $r = \mathsf{rule}(\rho, j, \rho')$. Let us call $h = (T, \mathsf{so}, \mathsf{wr})$, $h' = (T', \mathsf{so}', \mathsf{wr}')$ and e to $\mathsf{history}(\rho)$, $\mathsf{history}(\rho')$ and the last event in po-order belonging to $\mathtt{last}(h, j)$ respectively. By induction hypothesis, h is a full history. To conclude the inductive step, we show that for every possible rule r s.t. $r = \mathsf{rule}(\rho, j, \rho')$, the history h' is also a full history. In particular, by Definitions 2.2.1 and 4.2.1, it suffices to prove that $\mathsf{so}' \cup \mathsf{wr}'$ is an acyclic relation and that for every variable x and read event r, $\mathsf{wr}'_x{}^{-1}(r)\downarrow$ if and only if r does not read x from a local write and in such case, $\mathtt{value}_{\mathsf{wr}'}(\mathsf{wr}'_x{}^{-1}(r), x) \neq \bot$.

- LOCAL, IF-FALSE, IF-TRUE, INSERT, COMMIT, ABORT: The result trivially holds as $\mathsf{reads}(T') = \mathsf{reads}(T), \mathsf{wr}' = \mathsf{wr}$ and $\mathsf{so}' = \mathsf{so}$; using that h is consistent.
- BEGIN: We observe that $h' = h \oplus_j (e, \mathtt{begin})$, so $T = T \cup \{\mathtt{last}(h, j)\}$, $\mathsf{reads}(T') = \mathsf{reads}(T)$, $\mathsf{wr}' = \mathsf{wr}$ and $\mathsf{so}' = \mathsf{so} \cup \{(t', \mathtt{last}(h, j)) \mid \mathtt{ses}(t') = j\}$. In such case, by Lemma 4.3.4, t is $\mathsf{so}' \cup \mathsf{wr}'$-maximal. Thus, $\mathsf{so}' \cup \mathsf{wr}'$ is acyclic as $\mathsf{so} \cup \mathsf{wr}$ is also acyclic. Finally, as $\mathsf{wr}' = \mathsf{wr}$, we conclude that h' is a full history.
- SELECT, UPDATE, DELETE: Here $h' = (h \oplus_j (e, \mathsf{rule}(\rho, j, s))) \bigoplus_{x \in \mathsf{Keys}, \vec{\mathsf{w}}[x] \neq \bot} \mathsf{wr}(\vec{\mathsf{w}}[x], e)$ where e is the new event executed and $\vec{\mathsf{w}}$ is defined following the descriptions in Figures 4.5 and 4.6. In this case, $T' = T, \mathsf{reads}(T') = \mathsf{reads}(T) \cup \{e\}$, $\mathsf{so}' = \mathsf{so}$, $\forall x \in \mathsf{Keys}$ s.t. $\vec{\mathsf{w}}[x] = \bot$, $\mathsf{wr}'_x = \mathsf{wr}_x$ and $\forall x \in \mathsf{Keys}$ s.t. $\vec{\mathsf{w}}[x] \neq \bot$, $\mathsf{wr}'_x = \mathsf{wr}_x \cup \{(\vec{\mathsf{w}}[x], e)\}$. Note that as the timestamp of any event is always positive and $\vec{\mathsf{T}}(\mathtt{init}) = 0$; for any key x, $\mathsf{w}[x] \neq \bot$ if and only if $\mathtt{localWr}[x] = \bot$. Thus, $\vec{\mathsf{w}}$ is well defined, and $\mathsf{wr}'_x{}^{-1}(r)\downarrow$ if and only $\mathtt{localWr}[x] = \bot$. In such case, as any event w writes on a key x if and only if $\mathtt{value}_{\mathsf{wr}}(w, x) \neq \bot$, we conclude that $\mathtt{value}_{\mathsf{wr}'}(\mathsf{wr}'_x{}^{-1}(r), x) \neq \bot$. To conclude the result, we need to show that $\mathsf{so}' \cup \mathsf{wr}'$ is acyclic. As ρ is reachable, by Figure 4.6's definition we know that for any event r and key x, if $\mathsf{wr}_x^{-1}(r)\downarrow$, $\mathsf{tr}(\mathsf{wr}_x^{-1}(r)) \in \mathsf{cmtt}(h)$. Thus, by Lemma 4.3.4, $\mathtt{last}(h, j)$ is $\mathsf{so}' \cup \mathsf{wr}'$-maximal as it is not committed. Therefore, by the definition of so' and wr', as $\mathsf{so} \cup \mathsf{wr}$ is acyclic and $\mathtt{last}(h, j)$ is $\mathsf{so}' \cup \mathsf{wr}'$-maximal, $\mathsf{so}' \cup \mathsf{wr}'$ is also acyclic. In conclusion, h' is a full history.

□

Once proven that for any run ρ, $\mathsf{history}(\rho)$ is a full history, we need to prove that there exists a commit order co_ρ that witnesses $\mathsf{history}(\rho)$ consistency. Equation 4.1 defines a relation that we prove in Lemma 4.3.6 that it is a total order for $\mathsf{history}(\rho)$.

$$(t, t') \in \mathsf{co}_\rho \iff \begin{cases} t \in \mathsf{complete}(T) \land t' \in \mathsf{complete}(T) \land \vec{\mathsf{T}}(\mathtt{end}(t)) < \vec{\mathsf{T}}(\mathtt{end}(t')) \text{ or} \\ t \in \mathsf{complete}(T) \land t' \notin \mathsf{complete}(T) \text{ or} \\ t \notin \mathsf{complete}(T) \land t' \notin \mathsf{complete}(T) \land \vec{\mathsf{T}}(\mathtt{begin}(t)) < \vec{\mathsf{T}}(\mathtt{begin}(t')) \end{cases} \tag{4.1}$$

Lemma 4.3.6. *For every run ρ, the relation co_ρ defined above is a commit order for* $\mathsf{history}(\rho)$.

Proof. We prove by induction on the length of a run ρ that the relation co_ρ defined by the equation below is a commit order for $\mathsf{history}(\rho)$, i.e., if $\mathsf{history}(\rho) = (T, \mathsf{so}, \mathsf{wr})$, then $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{co}_\rho$.

The base case, where ρ is composed only by an initial configuration is immediate as in such case $\mathsf{wr} = \emptyset$. Let us suppose that for every run of length at most n the property holds and let ρ' a run of length $n + 1$. As ρ' is a sequence of configurations, there exist a reachable run ρ of length n, a session j and a rule r s.t. $\mathsf{r} = \mathsf{rule}(\rho, j, \rho')$. Let us call $h = (T, \mathsf{so}, \mathsf{wr})$,

$h' = (T', \mathsf{so}', \mathsf{wr}')$ and e to $\mathsf{history}(\rho)$, $\mathsf{history}(\rho')$ and the last event in po-order belonging to $\mathtt{last}(h, j)$ respectively. By induction hypothesis, co_ρ is a commit order for h. To conclude the inductive step, we show that $\mathsf{co}_{\rho'}$ is also a commit order for h'.

- LOCAL, IF-FALSE, IF-TRUE: As $h = h'$ and $\vec{\mathsf{T}}_{\rho'} = \vec{\mathsf{T}}_\rho$, $\mathsf{co}_{\rho'} = \mathsf{co}_\rho$. Thus, the result trivially holds.

- BEGIN: In this case, $e = \mathtt{begin}(\mathtt{last}(h, j))$ and $\mathtt{last}(h, j) \not\in \mathsf{complete}(T_{\rho'})$. Note that for any event $e' \neq e$, $\vec{\mathsf{T}}(e) > \vec{\mathsf{T}}(e')$ and $\mathsf{complete}(T'_\rho) = \mathsf{complete}(\rho)$. Thus, $\mathsf{co}_{\rho'} = \mathsf{co}_\rho \cup \{(t', \mathtt{last}(h', j)) \mid t' \in T\}$. As $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{co}_\rho$, $\mathsf{wr}' = \mathsf{wr}$ and $\mathsf{so}' = \mathsf{so} \cup \{(t', \mathtt{last}(h, j)) \mid \mathsf{ses}(t') = j\}$, $\mathsf{so}' \cup \mathsf{wr}' \subseteq \mathsf{co}_{\rho'}$; so $\mathsf{co}_{\rho'}$ is a commit order for h'.

- INSERT: In this case, as $\mathsf{complete}(T'_\rho) = \mathsf{complete}(T_\rho)$, $\mathsf{co}_{\rho'} = \mathsf{co}_\rho$. Hence, as $\mathsf{so}' = \mathsf{so}$ and $\mathsf{wr}' = \mathsf{wr}$, $\mathsf{so}' \cup \mathsf{wr}' \subseteq \mathsf{co}_{\rho'}$.

- SELECT, UPDATE, DELETE: Once again, as $\mathsf{complete}(T'_\rho) = \mathsf{complete}(T_\rho)$, $\mathsf{co}_{\rho'} = \mathsf{co}_\rho$. Note that $\mathsf{so}' = \mathsf{so}$, $\forall x \in \mathsf{Keys}$ s.t. $\vec{\mathsf{w}}[x] = \bot$, $\mathsf{wr}'_x = \mathsf{wr}_x$ and $\forall x \in \mathsf{Keys}$ s.t. $\vec{\mathsf{w}}[x] \neq \bot$, $\mathsf{wr}'_x = \mathsf{wr}_x \cup \{(\vec{\mathsf{w}}[x], e)\}$. In the latter case, where $\vec{\mathsf{w}}[x] \neq \bot$, we know that $\mathsf{tr}(\vec{\mathsf{w}}[x]) \in \mathsf{complete}(T)$ thanks to the definitions on Figure 4.6. By Equation 4.1, as $\mathtt{last}(h, j)$ is pending, we deduce that $(\mathsf{tr}(\vec{\mathsf{w}}[x]), \mathsf{tr}(e)) \in \mathsf{co}_{\rho'}$. Therefore, as $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{co}_\rho = \mathsf{co}_{\rho'}$, we conclude that $\mathsf{so}' \cup \mathsf{wr}' \subseteq \mathsf{co}_{\rho'}$.

- COMMIT, ABORT: In this case, $e = \mathtt{endlast}(h, j)$, $\mathsf{co}_{\rho'} \restriction_{T \setminus \{\mathtt{last}(h,j)\} \times T \setminus \{\mathtt{last}(h,j)\}} = \mathsf{co}_\rho \restriction_{T \setminus \{\mathtt{last}(h,j)\} \times T \setminus \{\mathtt{last}(h,j)\}}$, $\mathsf{so}' = \mathsf{so}$ and $\mathsf{wr}' = \mathsf{wr}$. Thus, to prove that $\mathsf{so}' \cup \mathsf{wr}' \subseteq \mathsf{co}_{\rho'}$ we only need to discuss about $\mathtt{last}(h, j)$. By Lemma 4.3.4, $\mathtt{last}(h, j)$ is $\mathsf{so}' \cup \mathsf{wr}'$-maximal. Hence, we focus on proving that for any transaction t' s.t. $(t', \mathtt{last}(h, j)) \in \mathsf{so}' \cup \mathsf{wr}'$, $(t', \mathtt{last}(h, j)) \in \mathsf{co}_{\rho'}$. Any such transaction t' must be completed by Lemma 4.3.4. However, by the definition on Figure 4.4, we know that $\vec{\mathsf{T}}(e) > \vec{\mathsf{T}}(\mathtt{end}(t'))$, so $(t', \mathtt{last}(h, j)) \in \mathsf{co}_{\rho'}$ by Equation 4.1. Thus, $\mathsf{so}' \cup \mathsf{wr}' \subseteq \mathsf{co}_{\rho'}$.

□

Lemma 4.3.7. *For every total run ρ, the $\mathsf{history}(\rho)$ is consistent.*

Proof. Let ρ^T be a total run. By Lemma 4.3.5, $\mathsf{history}(\rho^T)$ is a full history. Thus, to prove that $\mathsf{history}(\rho)$ is consistent, by Definition 4.3.1, we need to show that there exists a commit order co that witnesses its consistency. We prove by induction on the length of a prefix ρ of a total run ρ^T that the relation co_ρ defined in Equation 4.1 is a commit order that witnesses $\mathsf{history}(\rho)$'s consistency. Note that by Lemma 4.3.6, the relation co_ρ is indeed a commit order.

The base case, where ρ is composed only by an initial configuration is immediate as in such case $\mathsf{wr} = \emptyset$. Let us suppose that for every run of length at most n the property holds and let ρ' a run of length $n+1$. As ρ' is a sequence of configurations, there exist a reachable run ρ of length n, a session j and a rule r s.t. $\mathsf{r} = \mathsf{rule}(\rho, j, \rho')$. Let us call $h = (T, \mathsf{so}, \mathsf{wr})$, $h' = (T', \mathsf{so}', \mathsf{wr}')$ and e to $\mathsf{history}(\rho)$, $\mathsf{history}(\rho')$ and the last event in po-order belonging to $\mathtt{last}(h, j)$ respectively. By induction hypothesis, co_ρ is a commit order that witnesses

h's consistency. To conclude the inductive step, we show that for every possible rule r s.t. $\mathsf{r} = \mathsf{rule}(\rho, j, \rho')$, $\mathsf{co}_{\rho'}$ is a commit order witnessing h''s consistency.

By contradiction, let suppose that $\mathsf{co}_{\rho'}$ does not witness h''s consistency. Then, there exists a variable x, a read event r, an axiom $a \in \iota$ and two committed transactions t_1, t_2 s.t. $(t_1, e) \in \mathsf{wr}_x$, t_2 writes x, $\mathsf{vis}_a^{\mathsf{co}_{\rho'}}(t_2, r, x)$ holds in h' but $(t_1, t_2) \in \mathsf{co}_{\rho'}$; where $\iota = \vec{\mathsf{I}}(\mathtt{begin}(e))$. Thus, if we prove that such dependencies can be seen in h using co_ρ, we obtain a contradiction as co_ρ witnesses h's consistency. Note that as shown during the proof of Lemma 4.3.6, $\mathsf{co}_{\rho'} \restriction_{T \setminus \{\mathtt{last}(h,j)\} \times T \setminus \{\mathtt{last}(h,j)\}} = \mathsf{co}_\rho \restriction_{T \setminus \{\mathtt{last}(h,j)\} \times T \setminus \{\mathtt{last}(h,j)\}}$; so we simply prove that $\mathtt{last}(h, j)$ cannot be t_1, t_2, $\mathsf{tr}(r)$ or any intermediate transaction causing $\mathsf{vis}_a^{\mathsf{co}_{\rho'}}(t_2, r, x)$ to hold in h'.

- LOCAL, IF-FALSE, IF-TRUE: As $h = h'$ and $\mathsf{co}_{\rho'} = \mathsf{co}_\rho$, this case is impossible.
- BEGIN: In this case, $\mathsf{co}_{\rho'} = \mathsf{co}_\rho \cup \{(t', \mathtt{last}(h', j)) \mid t' \in T\}$. By Lemma 4.3.6, $\mathtt{last}(h', j)$ is $(\mathsf{so}' \cup \mathsf{wr}')$-maximal, so $\mathtt{last}(h', j) \neq t_1$. Moreover, $\mathsf{reads}(\mathtt{last}(h', j)) = \emptyset$, so $r \neq \mathsf{reads}(\mathtt{last}(h', j))$. In addition, $\mathtt{last}(h, j) \neq t_2$ as $\mathsf{writes}(\mathtt{last}(h, j)) = \emptyset$.
  - $\underline{a = \mathsf{Serializability}, \mathsf{Prefix} \text{ or } \mathsf{Read\ Committed}}$: In all cases, the axioms do not relate any other transactions besides t_1, t_2 and $\mathsf{tr}(r)$, so this case is impossible.
  - $\underline{a = \mathsf{Conflict}}$: In this case, $\mathtt{last}(h, j) \neq t_4$ as it is $\mathsf{co}_{\rho'}$-maximal; so this case is also impossible.
- INSERT: In this case, $\mathsf{co}_{\rho'} = \mathsf{co}_\rho$. Moreover, $\mathsf{reads}(T') = \mathsf{reads}(T)$, $\mathsf{writes}(T') = \mathsf{writes}(T)$, $\mathsf{so}' = \mathsf{so}$ and $\mathsf{wr}' = \mathsf{wr}$. Thus, this case is also impossible.
- SELECT, UPDATE, DELETE: In this case, $\mathsf{co}_{\rho'} = \mathsf{co}_\rho$, $\mathsf{so}' = \mathsf{so}$, $\forall x \in \mathsf{Keys}$ s.t. $\vec{\mathsf{w}}[x] = \bot$, $\mathsf{wr}'_x = \mathsf{wr}_x$ and $\forall x \in \mathsf{Keys}$ s.t. $\vec{\mathsf{w}}[x] \neq \bot$, $\mathsf{wr}'_x = \mathsf{wr}_x \cup \{(\vec{\mathsf{w}}[x], e)\}$. As $\mathtt{last}(h, j)$ is pending, by Lemma 4.3.6, $\mathtt{last}(h, j) \neq t_1$ as it is $(\mathsf{so}' \cup \mathsf{wr}')$-maximal. Moreover, as $\mathsf{writes}(\mathtt{last}(h, j)) = \emptyset$, $\mathtt{last}(h, j) \neq t_2$. Then, we analyze if $\mathtt{last}(h, j)$ can be $\mathsf{tr}(r)$ (and thus, $r = e$) or any intermediate transaction. Note that for all three isolation levels we study, $\mathsf{readFrom}$ returns the value written by the transaction with the last commit timestamp for a given snapshot time. Hence, as $(t_1, r) \in \mathsf{wr}_x$ and $(t_2, \mathsf{tr}(r)) \in \mathsf{co}_\rho$, we deduce that $\vec{\mathsf{T}}_\rho(\mathtt{commit}(t_2)) > \vec{\mathsf{T}}_\rho(\mathtt{begin}(\mathtt{last}(h, j)))$. We continue the analysis distinguishing between one case per axiom:
  - $\underline{a = \mathsf{Serializability}}$: As ρ' is a prefix of a total run ρ^T, there exists runs $\hat{\rho}, \hat{\rho}'$ s.t. $\mathsf{rule}(\hat{\rho}, j', \hat{\rho}')$ is either COMMIT or ABORT and both a prefix of ρ^T; where j' is the session of $\mathsf{tr}(r)$. Without loss of generality, we can assume that $\hat{\rho}$ and $\hat{\rho}'$ have minimal size; so $\mathtt{last}(\mathsf{history}(\hat{\rho}), j') = \mathsf{tr}(r)$. As ρ^T is total and $\hat{\rho}'$ is a prefix of ρ^T, $\mathsf{validate}_\iota(\mathsf{history}(\hat{\rho}, \vec{\mathsf{T}}_{\hat{\rho}'}, \mathsf{tr}(r)))$ holds.

    By the monotonicity of $\vec{\mathsf{T}}$, $\vec{\mathsf{T}}_{\rho'} \subseteq \vec{\mathsf{T}}_{\hat{\rho}'}$. Hence, as $(t_1, r) \in \mathsf{wr}_x$ and $\vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{commit}(t_1)) < \vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{commit}(t_2))$, by the definitions of Figure 4.5 and Figure 4.6 we deduce that $\vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{begin}(\mathsf{tr}(r))) < \vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{commit}(t_2))$. However, as $\vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{begin}(\mathsf{tr}(r))) < \vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{commit}(t_2)) < \vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{end}(\mathsf{tr}(r)))$, $\mathsf{tr}(r)$ reads x, t_2 writes x; we conclude that $\mathsf{validate}_{\mathsf{SER}}(\mathsf{history}(\hat{\rho}'), \vec{\mathsf{T}}_{\hat{\rho}'}, \mathsf{tr}(r))$ does not hold; so this case is impossible.

- $\underline{a = \mathsf{Conflict}}$: In this case, $\mathtt{last}(h, j)$ cannot be an intermediate transaction nor $\mathsf{tr}(r)$ as $\mathsf{writes}(\mathtt{last}(h, j)) = \emptyset$; so this case is also impossible.
- $\underline{a = \mathsf{Prefix}}$: In this case, $\mathtt{last}(h, j)$ cannot be an intermediate transaction as by Lemma 4.3.4, $\mathtt{last}(h, j)$ is $\mathsf{so}' \cup \mathsf{wr}'$-maximal. Thus, $\mathtt{last}(h, j)$ must be $\mathsf{tr}(r)$ and $e = r$. Therefore, there exists a transaction t_4 s.t. $(t_2, t_4) \in {\mathsf{co}_{\rho'}}^*$ and $(t_4, \mathtt{last}(h, j)) \in (\mathsf{so}' \cup \mathsf{wr}')$. Note that t_4 must be committed and that $\vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_4)) < \vec{\mathsf{T}}_{\rho'}(\mathtt{begin}(\mathtt{last}(h, j)))$. Hence, as $(t_2, t_4) \in {\mathsf{co}_{\rho'}}^*$ and $(t_1, t_2) \in \mathsf{co}_{\rho'}$ and they are both committed, we deduce that $\vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_2)) < \vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_4)) < \vec{\mathsf{T}}_{\rho'}(\mathtt{begin}(\mathtt{last}(h, j)))$. However, this contradicts that $\vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_2)) > \vec{\mathsf{T}}_{\rho'}(\mathtt{begin}(\mathtt{last}(h, j)))$ Thus, this case is impossible.
- $\underline{a = \mathsf{Read\ Committed}}$: In this case, $\mathtt{last}(h, j)$ must be $\mathsf{tr}(r)$ and in particular, $e = r$. As depicted on Figure 4.5 and Figure 4.6, as $(t_1, r) \in \mathsf{wr}_x$, $\vec{\mathsf{S}}_{\rho'}(e) \leq \vec{\mathsf{T}}_{\rho'}(t_1)$. However, as $(t_1, t_2) \in \mathsf{co}_{\rho'}$, $\vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_1)) < \vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_2))$. Hence, as $(t_2, e) \in (\mathsf{so} \cup \mathsf{wr}); \mathsf{po}^*$, there exists an event $e' \in \mathtt{last}(h, j)$ s.t. $(e, e') \in \mathsf{po}^*$ and $\vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_2)) < \vec{\mathsf{T}}_{\rho'}(e')$. However, by $\mathsf{snapshot}_{\mathsf{RC}}$'s definition, $\vec{\mathsf{S}}(e') \leq \vec{\mathsf{S}}_{\rho'}(e)$; so we deduce that $\vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_1)) < \vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_2)) < \vec{\mathsf{S}}_{\rho'}(e)$. This contradicts the definition of $\mathsf{readFrom}$; so this case is impossible.

- COMMIT, ABORT: In this case, $\mathsf{co}_{\rho'} \restriction (T \setminus \{\mathtt{last}(h, j)\} \times T \setminus \{\mathtt{last}(h, j)\}) = \mathsf{co}_{\rho} \restriction (T \setminus \{\mathtt{last}(h, j)\} \times T \setminus \{\mathtt{last}(h, j)\})$, $\mathsf{so}' = \mathsf{so}$, $\mathsf{wr}' = \mathsf{wr}$. First, using that by induction hypothesis any prefix $\tilde{\rho}$ of ρ is consistent using $\mathsf{co}_{\tilde{\rho}}$; we define $\tilde{\rho}$ the prefix of ρ that introduces the read event r. As $\mathsf{history}(\tilde{\rho}) = (\tilde{T}, \tilde{\mathsf{so}}, \tilde{\mathsf{wr}})$ is consistent and $(t_1, r) \in \tilde{\mathsf{wr}}_x$; t_1 is committed. Hence, by the definitions of $\mathsf{readFrom}$ and $\mathsf{snapshot}_\iota$ on Figure 4.6 and the rules semantics on Figure 4.5, we deduce that $\vec{\mathsf{T}}_{\rho}(\mathtt{commit}(t_1)) > \vec{\mathsf{T}}_{\rho}(\mathtt{begin}(t_2))$. Next, as $\mathtt{last}(h, j)$ is pending in h, it is $\mathsf{so} \cup \mathsf{wr}$-maximal. Therefore, it is also $\mathsf{so}' \cup \mathsf{wr}'$-maximal; so it cannot play the role of t_1. However, it can play the role of t_2, $\mathtt{last}(h, j)$ or the role of an intermediate transaction. Let us analyze case by case depending on the axiom:

  - $\underline{a = \mathsf{Serializability}}$: Two sub-cases arise:
    * $\underline{\mathtt{last}(h, j) = t_2}$: I this case, t_2 writes x must hold. As ρ' is a prefix of a total run ρ^T, there exists runs $\hat{\rho}, \hat{\rho}'$ s.t. $\mathsf{rule}(\hat{\rho}, j', \hat{\rho}')$ is either COMMIT or ABORT and both a prefix of ρ^T; where j' is the session of $\mathsf{tr}(r)$. Without loss of generality, we can assume that $\hat{\rho}$ and $\hat{\rho}'$ have minimal size; so $\mathtt{last}(\mathsf{history}(\hat{\rho}), j') = \mathsf{tr}(r)$. As ρ^T is total and $\hat{\rho}'$ is a prefix of ρ^T, $\mathsf{validate}_\iota(\mathsf{history}(\hat{\rho}, \vec{\mathsf{T}}_{\hat{\rho}'}, \mathsf{tr}(r)))$ holds. Note that as $(t_1, t_2) \in \mathsf{co}_{\rho'}$ and they are both committed, $\vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{commit}(t_1)) < \vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{commit}(t_2))$. However, $\mathsf{tr}(r)$ reads x, t_2 writes x and $\vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{begin}(\mathsf{tr}(r))) < \vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{commit}(\mathtt{last}(h, j))) < \vec{\mathsf{T}}_{\hat{\rho}'}(\mathtt{commit}(\mathsf{tr}(r)))$; which contradicts that $\mathsf{validate}_\iota(\mathsf{history}(\hat{\rho}, \vec{\mathsf{T}}_{\hat{\rho}'}, \mathsf{tr}(r)))$ holds. In conclusion, this case is impossible.
    * $\underline{\mathtt{last}(h, j) = \mathsf{tr}(r)}$: In such case, as t_1 and t_2 are committed, $(t_2, \mathtt{last}(h, j)) \in \mathsf{co}_{\rho}$ and $(t_1, t_2) \in \mathsf{co}_{\rho}$. Hence, this case is also impossible as co_{ρ} witnesses that h is consistent.

- $\underline{a = \mathsf{Prefix}}$: In this case, there exists a transaction t_4 s.t. $(t_2, t_4) \in \mathsf{co}_{\rho'}^*$ and $(t_4, \mathsf{tr}(r)) \in \mathsf{so}' \cup \mathsf{wr}'$. As $\mathtt{last}(h, j)$ is pending in h, by Lemma 4.3.4, $(\mathsf{so} \cup \mathsf{wr})$-maximal. Thus, as $\mathsf{so}' = \mathsf{so}$ and $\mathsf{wr}' = \mathsf{wr}$, $t_4 \neq \mathtt{last}(h, j)$. Moreover, as $(t_2, t_4) \in {\mathsf{co}_{\rho'}}^*$, t_4 is committed and $\mathtt{last}(h, j) \neq t_4$ is the $\mathsf{co}_{\rho'}$-maximal transaction that is committed, $t_2 \neq \mathtt{last}(h, j)$. Hence, $\mathtt{last}(h, j) = \mathsf{tr}(r)$. However, as $\mathsf{so}' = \mathsf{so}$, $\mathsf{wr}' = \mathsf{wr}'$ and $\mathsf{co}_{\rho'} \restriction_{T \setminus \{\mathtt{last}(h,j)\} \times T \setminus \{\mathtt{last}(h,j)\}} = \mathsf{co}_{\rho} \restriction_{T \setminus \{\mathtt{last}(h,j)\} \times T \setminus \{\mathtt{last}(h,j)\}}$; we conclude that $(t_1, t_2) \in \mathsf{co}_\rho$, $(t_2, t_4) \in {\mathsf{co}_\rho}^*$ and $(t_4, \mathtt{last}(h, j)) \in \mathsf{so} \cup \mathsf{wr}$; which contradicts that co_ρ witnesses h's consistency, so this case is impossible.
- $\underline{a = \mathsf{Conflict}}$: In this case, there exists a variable y and a transaction t_4 s.t. t_4 writes y, $\mathsf{tr}(r)$ writes y $(t_2, t_4) \in \mathsf{co}_{\rho'}^*$, $(t_4, \mathsf{tr}(r)) \in \mathsf{co}_{\rho'}$. As $\mathtt{last}(h, j)$ is the $\mathsf{co}_{\rho'}$-maximal transaction that is committed, $(t_2, \mathsf{tr}(r)), (t_4, \mathsf{tr}(r)) \in \mathsf{co}_{\rho'}$ and $\mathsf{writes}(\mathsf{tr}(r)) \neq \emptyset$, we deduce that $\mathtt{last}(h, j) \neq t_2, t_4$. Hence, $\mathtt{last}(h, j)$ must be $\mathsf{tr}(r)$ and $e = \mathtt{commit}(\mathtt{last}(h, j))$. On one hand, we observe that as $(t_4, \mathtt{last}(h, j)) \in \mathsf{co}_{\rho'}$ and they are both committed, $\vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_4)) < \vec{\mathsf{T}}_{\rho'}(e)$. On the other hand, as $(t_2, t_4) \in \mathsf{co}_{\rho'}^*$ and $\vec{\mathsf{T}}_{\rho'}(\mathtt{begin}(\mathsf{tr}(r))) < \vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_2))$; we conclude that $\vec{\mathsf{T}}_{\rho'}(\mathtt{begin}(\mathsf{tr}(r))) < \vec{\mathsf{T}}_{\rho'}(\mathtt{commit}(t_4))$. In conclusion, we obtain that $\mathsf{validate}_{\mathtt{SI}}(h', \vec{\mathsf{T}}_{\rho'}, \mathtt{last}(h, j))$ does not hold due to the existence of t_4; which contradict the hypothesis, so this case is impossible.
- $\underline{a = \mathsf{Read\ Committed}}$: In this case, $r \neq e$ as r is a read event and e is not, and $(t_2, r) \in (\mathsf{so}' \cup \mathsf{wr}'); \mathsf{po}'^*$. Hence, as $\mathsf{so}' = \mathsf{so}, \mathsf{wr}' = \mathsf{wr}$ and $\mathsf{po}' = \mathsf{po} \cup \{(e', e) \mid e' \in \mathtt{last}(h, j)\}$; $(t_2, r) \in (\mathsf{so} \cup \mathsf{wr}); \mathsf{po}^*$. Finally, as $\mathtt{last}(h, j)$ is pending in h, $\mathtt{last}(h, j) \neq t_2$. Thus, as $\mathsf{co}_{\rho'} \restriction_{T \setminus \{\mathtt{last}(h,j)\} \times T \setminus \{\mathtt{last}(h,j)\}} = \mathsf{co}_{\rho} \restriction_{T \setminus \{\mathtt{last}(h,j)\} \times T \setminus \{\mathtt{last}(h,j)\}}$; we deduce that $(t_1, t_2) \in \mathsf{co}_\rho$. However, this contradicts that co_ρ witnesses h's consistency; so this case is also impossible.

As every possible case is impossible, we deduce that the hypothesis, $\mathsf{co}_{\rho'}$ does not witnesses h''s consistency is false; so we conclude the proof of the inductive step.

□

4.4 Complexity of Checking Consistency

4.4.1 Saturation and Boundedness

We investigate the complexity of checking if a history is consistent. Our axiomatic framework characterize isolation levels as a conjunction of axioms as in Equation (2.1). However, some isolation levels impose stronger constraints than others. For studying the complexity of checking consistency, we classify them in two categories, saturable or not. An isolation level is *saturable* if its visibility relations are defined without using the co relation (i.e. the grammar in Equation (2.3) omits the co relation). Otherwise, we say that the isolation level is *non-saturable.* For example, $\mathtt{RC}$ and $\mathtt{RA}$ are saturable while $\mathtt{PC}, \mathtt{SI}$ and $\mathtt{SER}$ are not.

Definition 4.4.1. *An isolation configuration* $\mathsf{iso}(h)$ *is* saturable *if for every transaction* t, $\mathsf{iso}(h)(t)$ *is a saturable isolation level. Otherwise,* $\mathsf{iso}(h)$ *is* non-saturable.

We say an isolation configuration $\mathsf{iso}(h)$ is *bounded* if there exists a fixed $k \in \mathbb{N}$ s.t. for every transaction t, $\mathsf{iso}(h)(t)$ is defined as a conjunction of at most k axioms that contain at most k quantifiers. For example, `SER` employs one axiom and four quantifiers while `SI` employs two axioms called Prefix and Conflict with four and five quantifiers respectively. Any isolation configuration composed with `SER`, `SI`, `PC`, `RA` and `RC` isolation levels is bounded. We assume in the following that isolation configurations are bounded.

Checking consistency requires computing the $\mathtt{value}_{\mathsf{wr}}$ function and thus, evaluating `WHERE` predicates. In the following, we assume that evaluating `WHERE` predicates on a single row requires constant time.

4.4.2 Checking Consistency of Full Histories

Algorithm 6 computes necessary conditions for the existence of a consistent execution $\xi = (h, \mathsf{co})$ for a history h. It calls SATURATE, defined in Algorithm 5, to compute a "*partial*" commit order relation pco that includes $(\mathsf{so} \cup \mathsf{wr})^+$ and any other dependency between transactions that can be deduced from the isolation configuration. A consistent execution exists iff this partial commit order is acyclic.

Algorithm 5 Extending an initial pco relation with necessary ordering constraints

```
1: function SATURATE($h = (T, \mathsf{so}, \mathsf{wr})$, $\mathsf{pco}$)            ▷ $\mathsf{pco}$ must be transitive.
2:     $\mathsf{pco}_{\mathtt{res}} \leftarrow \mathsf{pco}$
3:     for all $x \in \mathsf{Keys}$ do
4:         for all $r \in \mathsf{reads}(h), t_2 \neq \mathsf{tr}(r) \in T$ s.t. $t_2$ writes $x$ and $t_2 \neq \mathsf{tr}(\mathsf{wr}_x^{-1}(r))$ do
5:             $t_1 \leftarrow \mathsf{tr}(\mathsf{wr}_x^{-1}(r))$            ▷ $t_1$ is well defined as $h$ is a full history.
6:             for all $\mathsf{v} \in \mathsf{vis}(\mathsf{iso}(h)(\mathsf{tr}(r)))$ do
7:                 if $\mathsf{v}(t_2, r, x)$ then
8:                     $\mathsf{pco}_{\mathtt{res}} \leftarrow \mathsf{pco}_{\mathtt{res}} \cup \{(t_2, t_1)\}$
9:     return $\mathsf{pco}_{\mathtt{res}}$
```

Algorithm 5 decides if a relation co is a commit order witnessing consistency of the history (Lemma 4.4.2) and it runs in polynomial time (Lemma 4.4.4).

Lemma 4.4.2. *For any full history $h = (T, \mathsf{so}, \mathsf{wr})$, the execution $\xi = (h, \mathsf{co})$ is consistent if and only if $\mathsf{pco}_{\mathtt{res}} = \text{SATURATE}(h, \mathsf{co})$ is acyclic.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history, $\xi = (h, \mathsf{co})$ be an execution of h and $\mathsf{pco}_{\mathtt{res}} = \text{SATURATE}(h, \mathsf{co})$ be the relation obtained thanks to Algorithm 5.

$\underline{\implies}$ Let us suppose that ξ is consistent. As co is acyclic, it suffice to prove that $\mathsf{pco}_{\mathtt{res}} = \mathsf{co}$. By contradiction, let us suppose that $\mathsf{pco}_{\mathtt{res}} \neq \mathsf{co}$. As $\mathsf{co} \subseteq \mathsf{pco}_{\mathtt{res}}$ (line 2), there exists t_1, t_2 s.t. $(t_2, t_1) \in \mathsf{pco}_{\mathtt{res}} \setminus \mathsf{co}$. In such case, such tuple must be added in line 8. Hence, there exists $x \in \mathsf{Keys}, e \in \mathsf{reads}(h)$ and $v \in \mathsf{vis}(\mathsf{iso}(h)(\mathsf{tr}(r)))$ s.t. $t_1 = \mathsf{wr}_x^{-1}(r)$ and $\mathsf{vis}_a^{\mathsf{co}}(t_2, r, x)$ holds in h. As ξ is consistent, $(t_2, t_1) \in \mathsf{co}$; which is impossible. Hence, $\mathsf{pco}_{\mathtt{res}} = \mathsf{co}$.

$\underline{\impliedby}$ Let us suppose that $\mathsf{pco}_{\mathtt{res}}$ is acyclic. By contradiction, let us suppose that ξ is not consistent. Then, there exists an read event r s.t. $C^{\mathsf{co}}_{\mathsf{iso}(h)(\mathsf{tr}(r))}(r)$ does not hold. Hence, by Equation (2.1), there exists $v \in \mathsf{vis}(\mathsf{iso}(h)(\mathsf{tr}(r)))$, $x \in \mathsf{Keys}$, $t_2 \in T$ s.t. $\mathsf{v}(\mathsf{co})(t_2, r, x)$ hold in h but $(t_2, t_1) \notin \mathsf{co}$; where $t_1 = \mathsf{wr}_x^{-1}(r)$. In such case, Algorithm 5 ensures in line 8 that

$(t_2, t_1) \in \mathsf{pco}_{\mathtt{res}}$. However, as $\mathsf{co} \subseteq \mathsf{pco}_{\mathtt{res}}$ (line 2), co is a total order and $\mathsf{pco}_{\mathtt{res}}$ is acyclic, $\mathsf{co} = \mathsf{pco}_{\mathtt{res}}$. Thus, $(t_2, t_1) \in \mathsf{co}$; which is impossible. Thus, ξ is consistent. □

Lemma 4.4.3. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history s.t. $\mathsf{iso}(h)$ is bounded by $k \in \mathbb{N}$, $x \in \mathsf{Keys}$ be a key, $t \in T$ be a transaction, r be a read event, $\mathsf{pco} \subseteq T \times T$ be a partial order and v be a visibility relation in $\mathsf{vis}(\mathsf{iso}(h)(\mathsf{tr}(r)))$. Evaluating $\mathsf{v}(\mathsf{pco})(t, r, x)$ is in $\mathcal{O}(|h|^{k-2})$.*

Proof. As $\mathsf{iso}(h)$ is bounded, there exists $k \in \mathbb{N}$ s.t. $|\mathsf{vis}(\mathsf{iso}(h)(t))| \leq k$. Hence, the number of quantifiers employed by a visibility relation is at most k (and at least 3 according to Equation 2.1). In addition, for each $\mathsf{v} \in \mathsf{vis}(\mathsf{iso}(h)(t))$ evaluating each condition $\mathsf{v}(\mathsf{pco})(t, r, x)$ can be modelled with an algorithm that employ $k-3$ nested loops, one per existential quantifier employed by v, and that for each quantifier assignment evaluates the quantifier-free part of the formula.

First, we observe that as WrCons predicate only query information about the $k-1$ quantified events, the size of such sub-formula is in $\mathcal{O}(k)$. Next, we notice that as `WHERE` predicate can be evaluated in constant time, for every key x and event w, computing $\mathtt{value}_{\mathsf{wr}}(x, w)$ is in $\mathcal{O}(k \cdot T)$. Hence, as k is constant, evaluating the quantifier-free formula of v is in $\mathcal{O}(|h|)$ and thus, evaluating $\mathsf{v}(\mathsf{pco})(t, r, x)$ is in $\mathcal{O}(|h|^{k-3} \cdot |h|) = \mathcal{O}(|h|^{k-2})$. □

Lemma 4.4.4. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a full history, k be a bound on $\mathsf{iso}(h)$ and $\mathsf{pco} \subseteq T \times T$ be a partial order. Algorithm 5 runs in $\mathcal{O}(|h|^{k+1})$.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a full history. Algorithm 5 can be decomposed in two blocks: lines 4-8 and lines 6-8. Hence, the cost of Algorithm 5 is in $\mathcal{O}(|\mathsf{Keys}| \cdot |\mathsf{events}(h)| \cdot |T| \cdot U)$; where U is an upper-bound of the cost of evaluating lines 6-8. On one hand, both $|\mathsf{Keys}|$, $|\mathsf{events}(h)|$ and $|T|$ are in $\mathcal{O}(|h|)$. On the other hand, as $\mathsf{iso}(h)$ is bounded by k, by Lemma 4.4.3, $U \in \mathcal{O}(|h|^{k-2})$. Altogether, we deduce that Algorithm 5 runs in $\mathcal{O}(|h|^{k+1})$. □

Algorithm 6 Checking saturable consistency

```
1: function CHECKSATURABLE($h = (T, \mathsf{so}, \mathsf{wr})$)
2:     if $\mathsf{so} \cup \mathsf{wr}$ is cyclic then return false
3:     $\mathsf{pco} \leftarrow$ SATURATE($h, (\mathsf{so} \cup \mathsf{wr})^+$)
4:     return true if $\mathsf{pco}$ is acyclic, and false, otherwise
```

Algorithm 6 generalizes the results in [29] for full histories with heterogeneous saturable isolation configurations.

Theorem 4.4.5. *Checking consistency of full histories with saturable isolation configurations can be done in polynomial time.*

We split the proof of Theorem 4.4.5 in two Lemmas: Lemma 4.4.6 that proves the correctness of Algorithm 6 and Lemma 4.4.7 that ensures its polynomial-time behavior.

Lemma 4.4.6. *For every full history $h = (T, \mathsf{so}, \mathsf{wr})$ whose isolation configuration is saturable, Algorithm 6 returns* true *if and only if h is consistent.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ a full history whose isolation configuration is saturable and let pco be the visibility relation defined in line 3 in Algorithm 6.

On one hand, let suppose that h is consistent and let $\xi = (h, \mathsf{co})$ be a consistent execution of h. If we show that $\mathsf{pco} \subseteq \mathsf{co}$, we can conclude that Algorithm 6 returns true as co is acyclic. Let $(t_2, t_1) \in \mathsf{pco}$ and let us prove that $(t_2, t_1) \in \mathsf{co}$. As $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{co}$, by the definition of commit order, we can assume that $(t_2, t_1) \in \mathsf{pco} \setminus (\mathsf{so} \cup \mathsf{wr})$. In such case, there must exists $x \in \mathsf{Keys}, e \in \mathsf{reads}(h)$ and $\mathsf{v} \in \mathsf{vis}(\mathsf{iso}(h)(\mathsf{tr}(e)))$ s.t. t_2 writes x and $\mathsf{v}((\mathsf{so} \cup \mathsf{wr})^+)(t_2, e, x)$ holds. As $\mathsf{iso}(h)(\mathsf{tr}(e))$ is saturable, $\mathsf{v}((\mathsf{so} \cup \mathsf{wr})^+)(t_2, e, x)$ holds. Hence, as co is a commit order and $(\mathsf{so} \cup \mathsf{wr})^+ \subseteq \mathsf{co}$; $\mathsf{v}(\mathsf{co})(t_2, e, x)$ also holds. Therefore, as co witnesses h's consistency, we deduce that $(t_2, t_1) \in \mathsf{co}$.

On the other hand, let us suppose that Algorithm 6 returns true. Then, pco must be acyclic by the condition in line 4. Therefore, as pco is acyclic it can be extended to a total order co. Let us prove that the execution $\xi = (h, \mathsf{co})$ is consistent. Let $x \in \mathsf{Keys}, t_2 \in T, e \in \mathsf{reads}(h)$ and $\mathsf{v} \in \mathsf{vis}(\mathsf{iso}(h)(\mathsf{tr}(e)))$ s.t. t_2 writes x and $\mathsf{v}(\mathsf{co})(t_2, e, x)$ holds. As Algorithm 6 returns true, we deduce that Algorithm 5 checks the condition at line 7. As $\mathsf{iso}(h)(\mathsf{tr}(e))$ is saturable, $\mathsf{v}((\mathsf{so} \cup \mathsf{wr})^+)(t_2, e, x)$ also holds. Thus, $(t_2, t_1) \in \mathsf{pco}$. As $\mathsf{pco} \subseteq \mathsf{co}$, $(t_2, t_1) \in \mathsf{co}$; so co witnesses h's consistency.

$\square$

Lemma 4.4.7. *For every full history h whose isolation configuration is bounded, Algorithm 6 runs in polynomial time with respect $\mathcal{O}(|h|)$.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a full history whose isolation configuration is saturable. First, we observe that checking if a graph $G = (V, E)$ is acyclic can be easily done with a DFS in $\mathcal{O}(|V| + |E|)$. Thus, the cost of checking acyclicity of both $\mathsf{so} \cup \mathsf{wr}$ (line 2) and pco (line 4) is in $\mathcal{O}(|T| + |T|^2) = \mathcal{O}(|T|^2) \subseteq \mathcal{O}(|h|^2)$. Furthermore, by Lemma 4.4.4, the cost of executing Algorithm 5 is in $\mathcal{O}(|h|^{k+1})$; where k is a bound in $\mathsf{iso}(h)$. Thus, checking h's consistency with Algorithm 6 can be done in polynomial time. $\square$

For bounded non-saturable isolation configurations, checking if a history is consistent is NP-complete as an immediate consequence of the results in [29]. These previous results apply to the particular case of transactions having the same isolation level and being formed of classic read and write instructions on a fixed set of variables (i.e. transactions defined as in Chapter 3). The latter can be simulated by SQL queries using `WHERE` predicates for selecting rows based on their key being equal to some particular value. For instance, $\texttt{SELECT}(\lambda r : \mathsf{key}(r) = x)$ simulates a read of a "variable" x.

4.4.3 Checking Consistency of Client Histories

We show that going from full histories to client histories, the consistency checking problem becomes NP-complete, independently of the isolation configurations. Intuitively, NP-hardness comes from keys that are not included in outputs of SQL queries. The justification for the consistency of omitting such rows can be ambiguous, e.g., multiple values written to a row may not satisfy the predicate of the `WHERE` clause, or multiple deletes can justify the absence of a row.

The *width* of a history $\texttt{width}(h)$ is the maximum number of transactions which are pairwise incomparable w.r.t. so. In a different context, previous work [29] showed that bounding the width of a history (consider it to be a constant) is a sufficient condition for obtaining polynomial-time consistency checking algorithms. This is not true for client histories.

Theorem 4.4.8. *Checking consistency of bounded-width client histories with bounded isolation configuration stronger than* $\texttt{RC}$ *and* $\texttt{width}(h) \geq 3$ *is NP-complete.*

The proof of NP-hardness uses a reduction from 1-in-3 SAT which is inspired by the work of Gibbons and Korach [58] (Theorem 2.7) concerning sequential consistency for shared memory implementations. Our reduction is a non-trivial extension because it has to deal with any weak isolation configuration stronger than $\texttt{RC}$.

The proof of Theorem 4.4.8 is structured in two parts: proving that the problem is in NP and proving that is NP-hard. The first part corresponds to Lemma 4.4.9; which is analogous as the proof of Lemma 4.4.21. The second part, based on a reduction to 1-in-3 SAT problem, corresponds to Lemmas 4.4.10, 4.4.12 and 4.4.18.

Lemma 4.4.9. *The problem of checking consistency for a bounded width client history h with an isolation configuration stronger than* $\texttt{RC}$ *and* $\texttt{width}(h) \geq 3$ *is in NP.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ a client history whose isolation configuration is stronger than $\texttt{RC}$. Guessing a witness of h, $\overline{h}$, and an execution of $\overline{h}$, $\xi = (\overline{h}, \mathsf{co})$, can be done in $\mathcal{O}(|\mathsf{Keys}| \cdot |\mathsf{events}(h)|^2 + |T|^2) \subseteq \mathcal{O}(|h|^3)$. By Lemma 4.4.2, checking if ξ is consistent is equivalent as checking if $\textsc{saturate}(h', \mathsf{co})$ is an acyclic relation. As by Lemma 4.4.4, Algorithm 5 requires polynomial time, we conclude the result. $\square$

For showing NP-hardness, we will reduce 1-in-3 SAT to checking consistency. Let φ be a boolean formula with n clauses and m variables of the form $\varphi = \bigwedge_{i=1}^{n} (v_i^0 \vee v_i^1 \vee v_i^2)$; we construct a history h_φ s.t. h_φ is consistent if and only if φ is satisfiable with exactly only one variable assigned the value $\texttt{true}$. The key idea is designing a history with width 3 that is stratified in *rounds*, one per clause. In each round, three transactions, one per variable in the clause, “compete” to be first in the commit order. The one that precedes the other two correspond to the variable in φ that is satisfied.

First, we define the round 0 corresponding to the variables of φ. For every variable $\mathrm{x}_i \in \texttt{var}(\varphi), 1 \leq i \leq m$ we define an homonymous key x_i that represents such variable. Doing an abuse of notation, we say that $x_i \in \texttt{var}(\varphi)$. Then, we create two transactions 1_i and 0_i associated to the two states of x_i, 1 and 0. The former contains the event $\texttt{INSERT}(\{x_i : 1, 1_i : 1\})$ while the latter $\texttt{INSERT}(\{x_i : 0, 0_i : 1\})$. Both 1_i and 0_i write also on a special key named 1_i and 0_i respectively to indicate on the database that they have committed.

Next, we define rounds $1 - n$ representing each clause in φ. For each clause $C_i := (v_i^0 \vee v_i^1 \vee v_i^2), 1 \leq i \leq n$, we define the round i. Round i is composed of three transactions: t_i^0, t_i^1 and t_i^2, representing the choice of the variable among v_i^0, v_i^1 and v_i^2 that is selected in the clause C_i. Transactions t_i^j write on keys v_i^j and $v_i^{j+1 \bmod 3}$ to preserve the structure of the clause C_i, as well on the special homonymous key t_i^j to indicate that such transaction has been executed; in a similar way as we did in the round 0. For that, we impose that transactions t_i^j are composed of an event $\texttt{SELECT}(\lambda x : \texttt{eq}(x, v_i^j, v_i^{j+1 \bmod 3}, v_i^{j+2 \bmod 3}))$ followed by an event $\texttt{INSERT}(\{v_i^j : 0, v_i^{j+1 \bmod 3} : 1, t_i^j : -1\})$.

The function $\texttt{eq} : \mathsf{Rows} \times \mathsf{Keys}^3 \to \{\mathsf{true}, \mathsf{false}\}$ is described in Equation (4.2) and assumes that Rows contains two distinct values 0 and 1 and that there is a predicate $\texttt{val} : \mathsf{Rows} \to \{0, 1\}$ that returns the value of a variable in the database. Intuitively, for any key r, if a, b, c correspond to the three variables in a clause C_i (possibly permuted), whenever $\neg\texttt{eq}(r, a, b, c)$ holds, we deduce that the value assigned at key a is 1 while on the other two keys the assigned value is 0. Moreover, whenever r refers to any of the special keys such as $0_i, 1_i$ or t_i^j, the predicate $\texttt{eq}(r, a, b, c)$ always holds.

$$\texttt{eq}(r, a, b, c) = \begin{cases} \texttt{val}(r) \neq 1 & \text{if } \mathsf{key}(r) = a \\ \texttt{val}(r) \neq 0 & \text{if } \mathsf{key}(r) = b \vee \mathsf{key}(r) = c \\ \mathsf{true} & \text{if } \mathsf{key}(r) \in \{t_i^j \mid 1 \leq i \leq n, 0 \leq j \leq 2\} \\ \mathsf{true} & \text{if } \mathsf{key}(r) \in \{1_i, 0_i \mid 1 \leq i \leq m\} \\ \mathsf{false} & \text{otherwise} \end{cases} \tag{4.2}$$

Finally, we add an initial transaction that writes on every key the value 1. For that, we assume that Keys contains only one key per variable used in φ as well as one key per aforementioned transaction. We denote by T the set of all described transactions as well as by $\texttt{round}(t)$ to the round a transaction $t \in T$ belongs to.

We describe the session order in the history h_φ using an auxiliary relation $\overline{\mathsf{so}}$. We establish that $(1_i, 1_j), (0_i, 0_j) \in \overline{\mathsf{so}}$ for any pair of indices $i, j, 1 \leq i < j \leq m$. We also enforce that $(t_i^j, t_{i+1}^j) \in \overline{\mathsf{so}}$, for every $1 \leq i \leq n, 0 \leq j, j' \leq 2$. Finally, we connect round 0 with round 1 by enforcing that $(1_m, t_1^0) \in \overline{\mathsf{so}}$ and $(0_m, t_1^1) \in \overline{\mathsf{so}}$. Then, we denote by so to the transitive closure of $\overline{\mathsf{so}}$. Note that so is a union of disjoint total orders, so it is acyclic.

For describing the write-read relation, we distinguish between two cases: keys associated to variables in φ or to a transaction in T. On one hand, for every key $x_i, 1 \leq i \leq m$, we define $\mathsf{wr}_{x_i} = \emptyset$. On the other hand, for every key x associated to a transaction t_x and every read event r in a transaction t, we impose that $(t_x, r) \in \mathsf{wr}_x$ if $\texttt{round}(t_x) < \texttt{round}(t)$ while otherwise we declare that $(\texttt{init}, r) \in \mathsf{wr}_x$. Then, we denote by $\mathsf{wr} = \bigcup_{x \in \mathsf{Keys}} \mathsf{wr}_x$ as well as by h_φ to the tuple $h_\varphi = (T, \mathsf{so}, \mathsf{wr})$. A full depiction of h_φ can be found in Figure 4.7.

We observe that imposing $\mathsf{wr}_x = \emptyset$ on every key $x \in \texttt{var}(\varphi)$ ensures that, for any witness of h_φ, $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$, if $(w, r) \in \overline{\mathsf{wr}}$, then $\texttt{WHERE}(r)(\texttt{value}_{\overline{\mathsf{wr}}}(w, x)) = 0$. In particular, this implies that each transaction t_i^j must read key v_i^j from a transaction that writes 1 as value while it also must read keys $v_i^{j+1 \bmod 3}$ and $v_i^{j+2 \bmod 3}$ from a transaction that writes 0 as value. Intuitively, this property shows that φ is well-encoded in h_φ.

The proof is divided in four steps: Lemma 4.4.10 proves that the h_φ is a polynomial-size transformation of φ, Lemma 4.4.11 proves that the h_φ is indeed a history and Lemmas 4.4.12 and 4.4.18 prove that h_φ is consistent if and only if φ is 1-in-3 satisfiable.

Lemma 4.4.10. *h_φ is a polynomial size transformation on the length of φ.*

Proof. If φ has n clauses and m variables, h_φ employs $6n + 2m + 1$ transactions. As $m \leq 3n$, $|T| \in \mathcal{O}(n)$. The number of variables, $|\mathsf{Keys}| = m + |T|$, so $|\mathsf{Keys}| \in \mathcal{O}(n)$. As every transaction

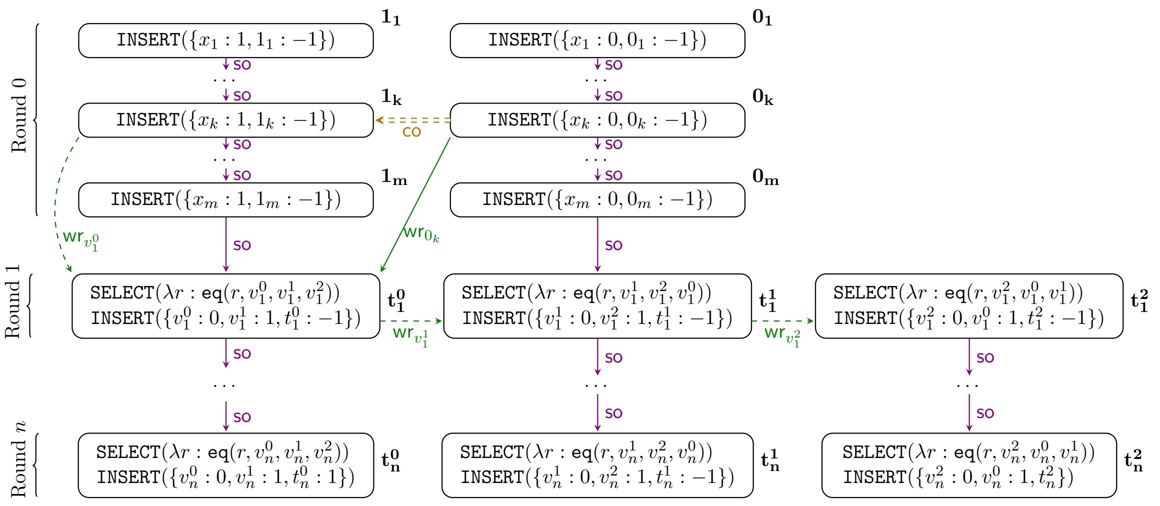

Round 0
Round 1
Round n
INSERT($\{x_1 : 1, 1_1 : -1\}$)
$\mathbf{1_1}$
INSERT($\{x_1 : 0, 0_1 : -1\}$)
$\mathbf{0_1}$
INSERT($\{x_k : 1, 1_k : -1\}$)
$\mathbf{1_k}$
INSERT($\{x_k : 0, 0_k : -1\}$)
$\mathbf{0_k}$
INSERT($\{x_m : 1, 1_m : -1\}$)
$\mathbf{1_m}$
INSERT($\{x_m : 0, 0_m : -1\}$)
$\mathbf{0_m}$
so
co
$\mathsf{wr}_{v_1^0}$
wr_{0_k}
SELECT($\lambda r : \mathsf{eq}(r, v_1^0, v_1^1, v_1^2)$) INSERT($\{v_1^0 : 0, v_1^1 : 1, t_1^0 : -1\}$)
$\mathbf{t_1^0}$
SELECT($\lambda r : \mathsf{eq}(r, v_1^1, v_1^2, v_1^0)$) INSERT($\{v_1^1 : 0, v_1^2 : 1, t_1^1 : -1\}$)
$\mathbf{t_1^1}$
SELECT($\lambda r : \mathsf{eq}(r, v_1^2, v_1^0, v_1^1)$) INSERT($\{v_1^2 : 0, v_1^0 : 1, t_1^2 : -1\}$)
$\mathbf{t_1^2}$
$\mathsf{wr}_{v_1^1}$
$\mathsf{wr}_{v_1^2}$
SELECT($\lambda r : \mathsf{eq}(r, v_n^0, v_n^1, v_n^2)$) INSERT($\{v_n^0 : 0, v_n^1 : 1, t_n^0 : 1\}$)
$\mathbf{t_n^0}$
SELECT($\lambda r : \mathsf{eq}(r, v_n^1, v_n^2, v_n^0)$) INSERT($\{v_n^1 : 0, v_n^2 : 1, t_n^1 : -1\}$)
$\mathbf{t_n^1}$
SELECT($\lambda r : \mathsf{eq}(r, v_n^2, v_n^0, v_n^1)$) INSERT($\{v_n^2 : 0, v_n^0 : 1, t_n^2\}$)
$\mathbf{t_n^2}$

Figure 4.7: Description of the history h_φ from Theorem 4.4.8. Dashed edges only belong to a possible consistent witness of h_φ, where we assume $v_1^0 = x_k$. Transaction t_1^0 reads v_1^0, v_1^1 and v_1^2 from round 0; imposing some constraints on the transactions that write them. Due to axiom RC's definition, transaction t_1^1 must read v_1^1 from t_1^0 while transaction t_1^2 must read v_1^1 from t_1^1.

has at most two events, $|\mathsf{events}(h_\varphi)| \in \mathcal{O}(n)$. Moreover, $\mathsf{wr} \subseteq \mathsf{Keys} \times T \times T$ and $\mathsf{so} \subseteq T \times T$, so $|\mathsf{wr}| \in \mathcal{O}(n^3)$ and $|\mathsf{so}| \in \mathcal{O}(n^2)$. Thus, h_φ is a polynomial transformation of φ. □

For proving that h_φ is a history, by Definition 2.2.1 it suffices to prove that $\mathsf{so} \cup \mathsf{wr}$ is an acyclic relation. Indeed, by our choice of wr, for every key x, wr_x^{-1} is a partial function that, whenever it is defined, associates reads to writes on x. Hence, from Lemma 4.4.11 we conclude that h_φ is a history.

Lemma 4.4.11. *The relation $\mathsf{so} \cup \mathsf{wr}$ is acyclic.*

Proof. For proving that $\mathsf{so} \cup \mathsf{wr}$ is acyclic, we reason by induction on the number of clauses. In particular, we show that for every pair of transactions t, t' if $\mathtt{round}(t') \leq i$ and $(t, t') \in \mathsf{so} \cup \mathsf{wr}$, then $\mathtt{round}(t) \leq i$ and $(t', t) \notin \mathsf{so} \cup \mathsf{wr}$.

- Base case: The base case refers to round 0; which contains `init` and transactions $0_j, 1_j, 1 \leq j \leq m$. We observe that transactions in round 0 do not contain any read event. Hence, $(t, t') \in \mathsf{so}$. In such case, the result immediately holds by construction of so.

- Inductive case: Let us suppose that the induction hypothesis holds for every $1 \leq i \leq k \leq n$ and let us prove it also for $k+1 \leq n$. If $\mathtt{round}(t') < k+1$, $\mathtt{round}(t') \leq k$ and the result holds by induction hypothesis; so we can assume without loss of generality that $\mathtt{round}(t') = k+1$. By construction of both so and wr, if $(t, t') \in \mathsf{so} \cup \mathsf{wr}$, $\mathtt{round}(t) < \mathtt{round}(t')$. Hence, $\mathtt{round}(t) \leq k$. By induction hypothesis on t, if $(t', t) \in \mathsf{so} \cup \mathsf{wr}$,

$\texttt{round}(t') \leq k < k+1 = \texttt{round}(t')$; which is impossible. Thus, we conclude that $(t', t) \not\in \mathsf{so} \cup \mathsf{wr}$.

□

Lemma 4.4.12. *If φ is* 1*-in-*3 *satisfiable then h_φ is consistent.*

Proof. Let $\alpha : \texttt{var}(\varphi) \to \{0,1\}$ an assignment that makes φ 1-in-3 satisfiable. To construct a witness of h_φ we define a write-read relation $\overline{\mathsf{wr}}$ that extends wr and a total order on its transactions. For that, we first define a total order co between the transactions in T. In Equation 4.2 we define two auxiliary relations $\hat{\mathtt{r}}$ and $\hat{\mathtt{b}}$ based on α that totally orders the transactions that belongs to the same round.

For every clause $C_i, 1 \leq i \leq n$ let j_i be the unique index s.t. $\alpha(v_i^{j_i}) = 1$; where $\alpha(v_i^{j_i})$ is a shortcut for $v_i^{j_i}[\alpha(\mathrm{x}_1)/\mathrm{x}_1 \ldots \alpha(\mathrm{x}_m)/\mathrm{x}_m]$. Such index allow us to order the transactions in the round i: $t_i^{j_i}$ preceding $t_i^{j_i+1 \bmod 3}$ while $t_i^{j_i+1 \bmod 3}$ preceding $t_i^{j_i+2 \bmod 3}$. Intuitively, $t_i^{j_i}$ must precede the other two transactions in the total order as v_i^j is the variable that is satisfied. Then, we connect every pair of consecutive rounds thanks to relation $\hat{\mathtt{c}}_1$.

For transactions in round 0, we enforce that transactions associated to the same variable are totally ordered using α. In particular, for every $i, 1 \leq i \leq m$, 0_i precedes 1_i in $\hat{\mathtt{b}}$ if and only if $\alpha(v_i) = 1$. Then, we connect every pair tuple in $\hat{\mathtt{b}}$ with relation $\hat{\mathtt{c}}_2$. Finally, we connect $\texttt{init}$ with transactions in round 0 as well as round 0 with round 1 thanks to relation $\hat{\mathtt{c}}_3$.

$$\begin{aligned}
\hat{\mathtt{r}} &= \left\{ \begin{array}{l} (t_i^{j_i}, t_i^{j_i+1 \bmod 3}) \\ (t_i^{j_i+1 \bmod 3}, t_i^{j_i+2 \bmod 3}) \end{array} \middle| \begin{array}{l} 1 \leq i \leq n, 0 \leq j_i \leq 2 \\ \alpha(v_i^{j_i}) = 1 \end{array} \right\} \\
\hat{\mathtt{b}} &= \{(0_i, 1_i) \mid x_i \in \mathsf{Keys} \wedge \alpha(x_i) = 1\} \cup \{(1_i, 0_i) \mid x_i \in \mathsf{Keys} \wedge \alpha(x_i) = 0\} \\
\hat{\mathtt{c}}_1 &= \{(t_i^{j_i+2 \bmod 3}, t_{i+1}^{j_{i+1}}) \mid 1 \leq i < n, 0 \leq j_i, j_{i+1} \leq 2, \alpha(v_i^{j_i}) = 1 = \alpha(v_i^{j_{i+1}})\} \\
\hat{\mathtt{c}}_2 &= \{(1_i, 0_j), (1_i, 1_j), (0_i, 0_j), (0_i, 1_j) \mid 1 \leq i < j \leq m\} \\
\hat{\mathtt{c}}_3 &= \{(\texttt{init}, 0_1), (\texttt{init}, 1_1)\} \cup \{(1_m, t_1^{j_1}), (0_m, t_1^{j_1}) \mid 0 \leq j_1 \leq 2, \alpha(v_i^{j_1}) = 1\}
\end{aligned} \tag{4.3}$$

Let $\mathsf{co} = (\hat{\mathtt{r}} \cup \hat{\mathtt{c}}_1 \cup \hat{\mathtt{b}} \cup \hat{\mathtt{c}}_2 \cup \hat{\mathtt{c}}_3)^+$. The proof of Lemma 4.4.12 concludes thanks to Lemmas 4.4.13 and 4.4.14, Proposition 4.4.16 and Corollary 4.4.17. First, Lemma 4.4.13 proves that the relation co is a total order between transactions. Then, Lemma 4.4.14 shows that co allow us define $\overline{h}$, a witness of h_φ. And finally, with the aid of Proposition 4.4.16 and Corollary 4.4.17 we conclude that $\overline{h}$ is consistent; so it is a consistent witness of h_φ.

□

Lemma 4.4.13. *The relation co is a total order.*

Proof. For proving that co is a total order, we show by induction that if $(t, t') \in \mathsf{co}$ and $\texttt{round}(t') \leq i$, then $\texttt{round}(t) \leq i$ and $(t', t) \not\in \mathsf{co}$.

- Base case: We observe that by construction of co, $t' \neq \texttt{init}$. We prove the base case by a second induction that if there exists $i', 1 \leq i' \leq m$ s.t. $t' \in \{0_{i'}, 1_{i'}\}$ and $(t, t') \in \mathsf{co}$ then either $t = \texttt{init}$ or there exists $i \leq i'$ s.t. $t \in \{0_i, 1_i\}$ and $(t', t) \not\in \mathsf{co}$.

- Base case: Let us suppose that $\alpha(x_0) = 1$ as the other case is symmetric. If $t = \texttt{init}$, $(t', t) \notin \mathsf{co}$ as $\texttt{init}$ is minimal in co. If not, then $t' = 1_1$ and $t = 0_1$. We conclude once more that $(t, t') \notin \mathsf{co}$ as 0_1 only have $\texttt{init}$ as a co-predecessor; which is co-minimal.
- Induction hypothesis: Let us suppose that the induction hypothesis holds for every $1 \leq i \leq k \leq m$ and let us prove it also for $k+1 \leq m$. If $i' < k$, we conclude the result by induction hypothesis; so we can assume that $i' = k$. Moreover, as $\texttt{init}$ is co-minimal, we can assume without loss of generality that $t \neq \texttt{init}$. Thus, by construction of co, there must exists $i, 1 \leq i \leq m$ s.t. $t \in \{0_i, 1_i\}$. In particular, $i \leq i'$. Thus, if $i < i'$ and (t', t) would be in co, by induction hypothesis on t we would deduce that $i' \leq i < i'$; which is impossible. Hence, we can assume that $i' = i$. Let us assume that $\alpha(x_i) = 1$ as the other case is symmetric. Thus, we deduce that $t = 0_i$ and $t' = 1_i$. We observe that $(t', t) \notin \hat{\mathtt{r}} \cup \hat{\mathtt{c}}_1 \cup \hat{\mathtt{b}} \cup \hat{\mathtt{c}}_2 \cup \hat{\mathtt{c}}_3$. As T is finite, if $(t', t) \in \mathsf{co}$, there would exist a transaction $t'' \neq t'$ s.t $(t'', t) \in \hat{\mathtt{r}} \cup \hat{\mathtt{c}}_1 \cup \hat{\mathtt{b}} \cup \hat{\mathtt{c}}_2 \cup \hat{\mathtt{c}}_3$ and $(t', t'') \in \mathsf{co}$. But in such case, either $t'' = \texttt{init}$ or there would exists an integer $i'', 1 \leq i'' < i \leq m$ s.t. $t'' \in \{0_{i''}, 1_{i''}\}$; which is impossible by induction hypothesis. In conclusion, $(t', t) \notin \mathsf{co}$.

- Inductive case: Let us suppose that the induction hypothesis holds for every $1 \leq i \leq k \leq n$ and let us prove it also for $k+1 \leq n$. Let thus t, t' transactions s.t. $\texttt{round}(t)' \leq k+1$ and $(t, t') \in \mathsf{co}$. If $\texttt{round}(t') < k+1$, $\texttt{round}(t') \leq k$ and the result holds by induction hypothesis; so we can assume without loss of generality that $\texttt{round}(t') = k+1$. By construction of co, $\texttt{round}(t) \leq k+1$. If $\texttt{round}(t) \leq k$ and $(t', t) \in \mathsf{co}$, by induction hypothesis on t we obtain that $\texttt{round}(t') \leq k < k+1 = \texttt{round}(t')$; which is impossible. Thus, we can also assume without loss of generality that $\texttt{round}(t) = k+1$. In such case, we observe that $\hat{\mathtt{c}}_1$, $\hat{\mathtt{b}}$ and $\hat{\mathtt{c}}_1$ do not order transactions belonging to the same round. Hence if $(t, t') \in \mathsf{co}$ and $(t', t) \in \mathsf{co}$, we deduce that $(t, t') \in \hat{\mathtt{r}}$ and $(t', t) \in \hat{\mathtt{r}}$. However, by construction of $\hat{\mathtt{r}}$, this is impossible, so we conclude once more that $(t', t) \notin \mathsf{co}$.

□

Next, we we construct a full history $\overline{h}$ using co that extends h_φ. For every key x and $\texttt{read}$ event r, we define w_x^r as follows:

$$w_x^r = \max_{\mathsf{co}}\{t \in T \mid t \text{ writes } x \wedge (t, r) \in \mathsf{co}\} \tag{4.4}$$

Observe that w_x^r is well-defined as co is a total order and $\texttt{init}$ write every key. For each key $x \in \texttt{var}(\varphi)$, we define the relation $\overline{\mathsf{wr}}_x = \{(w_x^r, r) \mid r \in \mathsf{reads}(h)\}$. Then, we define the relation $\overline{\mathsf{wr}} = \bigcup_{x \in \texttt{var}(\varphi)} \overline{\mathsf{wr}}_x \cup \mathsf{wr}$ as well as the history $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$. Lemma 4.4.14 proves that $\overline{h}$ is indeed a full history while Lemma 4.4.15 shows that $\overline{h}$ is a witness of h_φ.

Lemma 4.4.14. *$\overline{h}$ is a full history.*

Proof. For showing that $\overline{h}$ is a full history it suffices to show that $\mathsf{so} \cup \overline{\mathsf{wr}}$ is acyclic. As co is a total order and $\overline{\mathsf{wr}} \setminus \mathsf{wr} \subseteq \mathsf{co}$, proving that $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{co}$ concludes the result. First, we prove that $\mathsf{so} \subseteq \mathsf{co}$. Let t, t' be transactions s.t. $(t, t') \in \mathsf{so}$. In such case, $\mathtt{round}(t) \leq \mathtt{round}(t')$; and they only coincide if $\mathtt{round}(t) = \mathtt{round}(t') = 0$. Three cases arise:

- $\underline{\mathtt{round}(t) = \mathtt{round}(t') = 0}$: As $(t, t') \in \hat{\mathtt{c}}_2$, we conclude that $(t, t') \in \mathsf{co}$.
- $\underline{\mathtt{round}(t), \mathtt{round}(t') > 0}$: As $\mathtt{round}(t), \mathtt{round}(t') > 0$ and $\mathtt{round}(t) \leq \mathtt{round}(t')$, by construction of so we deduce that $\mathtt{round}(t) < \mathtt{round}(t')$. As co is transitive, we can assume without loss of generality that $\mathtt{round}(t') = \mathtt{round}(t) + 1$. Therefore, there exists $i, j, 1 \leq i < n, 0 \leq j \leq 2$ s.t. $t = t_i^j$ and $t' = t_{i+1}^j$. Let $j_i, j_{i+1}, 0 \leq j_i, j_{i+1} \leq 2$ be the integers s.t. $\alpha(v_i^{j_i}) = 1 = \alpha(v_{i+1}^{j_{i+1}})$. In such case, we know that $(t_i^j, t_i^{j_i+2 \bmod 3}) \in \hat{\mathtt{r}}^*$, $(t_i^{j_i+2 \bmod 3}, t_{i+1}^{j_i}) \in \hat{\mathtt{c}}_1$ and $(t_{i+1}^{j_{i+1}}, t_{i+1}^{j+1}) \in \hat{\mathtt{r}}^*$. Hence, as co is transitive, $(t, t') \in \mathsf{co}$.
- $\underline{\mathtt{round}(t) = 0, \mathtt{round}(t') > 0}$: In this case, as $\mathtt{round}(t) = 0$, there exists $i, 1 \leq i \leq m$ s.t. $x_i \in \mathtt{var}(\varphi), t \in \{0_i, 1_i\}$. We assume without loss of generality that $t = 1_i$ as the other case is symmetric. In addition, as $\mathtt{round}(t') > 0$ and $(t, t') \in \mathsf{so}$, there exists $i, 1 \leq i \leq n$ s.t. $t' = t_i^0$. We rely on the two previous proven cases to deduce the result: as $(0_i, 0_m) \in \mathsf{so} \subseteq \mathsf{co}$, $(0_m, t_0^{j_0}) \in \hat{\mathtt{c}}_3$, $(t_0^{j_0}, t_0^0) \in \hat{\mathtt{r}}^*$ and $(t_0^0, t_i^0) \in \mathsf{so} \subseteq \mathsf{co}$, we conclude that $(t, t') \in \mathsf{co}$.

Next, we prove that $\mathsf{wr} \subseteq \mathsf{co}$. Let r be a read event and w be a write event s.t. $(w, r) \in \mathsf{wr}$. Then, there exists $i, i', 1 \leq i < i' \leq n$ and $j, j', 0 \leq j, j' \leq 2$ s.t. $w = t_i^j$ and $\mathsf{tr}(r) = t_{i'}^{j'}$. Let $j_{i'-1}, j_{i'}, 0 \leq j_{i'-1}, j_{i'} \leq 2$ be the integers s.t. $\alpha(v_{i'-1}^{j_{i'-1}}) = 1 = \alpha(v_{i'}^{j'})$. In such case, we know that $(t_i^j, t_{i'-1}^j) \in \mathsf{so}^*$, $(t_{i'-1}^j, t_{i'-1}^{j_{i'}+2 \bmod 3}) \in \hat{\mathtt{r}}^*$, $(t_{i'-1}^{j_{i'}+2 \bmod 3}, t_{i'}^{j_i}) \in \hat{\mathtt{c}}_1$ and $(t_{i'}^{j_{i'}}, t_{i'}^{j'}) \in \hat{\mathtt{r}}^*$. As $\mathsf{so} \subseteq \mathsf{co}$ and co is transitive, we conclude that $(w, r) \in \mathsf{co}$. □

We show that $\overline{h}$ is indeed a full history, that is a witness of h_φ and that also witness h_φ's consistency.

Lemma 4.4.15. *The history $\overline{h}$ is a witness of h_φ.*

Proof. By Lemma 4.4.14 $\overline{h}$ is a full history. Hence, for proving that $\overline{h}$ is a witness of h_φ, we need to show that for every key $x \in \mathsf{Keys}$ and every read r, if $\mathsf{wr}_x^{-1}(r) \uparrow$, $\mathtt{WHERE}(r)(\mathtt{value}_{\overline{\mathsf{wr}}}(w_x^r, x)) = 0$. Note that by construction of h_φ, such cases are those where there exists an homonymous variable $\mathrm{x} \in \mathtt{var}(\varphi)$. In addition, we observe that if r is a read event, there exists indices $1 \leq i \leq n, 0 \leq j \leq 2$ s.t. $r \in t_i^j$. Thus, by Equation 4.2, we only need to prove that $\mathtt{WHERE}(r)(\mathtt{value}_{\overline{\mathsf{wr}}}(w_x^r, x)) = 0$ whenever x is v_i^0, v_i^1 or v_i^2.

We prove as an intermediate step that in each round, every key has the same value in $\overline{h}$. For every round i and key $x \in \mathtt{var}(\varphi)$, we consider the transaction $t_x^i = \max_{\mathsf{co}}\{t \mid t \text{ writes } x \wedge \mathtt{round}(t) \leq i\}$. We prove by induction on the number of the round that for every x associated with an homonymous variable x, $\mathtt{value}_{\overline{\mathsf{wr}}}(t_x^i, x) = \mathtt{value}_{\overline{\mathsf{wr}}}(t_x^0, x) = (x, \alpha(\mathrm{x}))$.

- $\underline{\text{Base case}}$: The base case, $i = 0$, is immediately trivial. Note that in this case, $\mathtt{value}_{\overline{\mathsf{wr}}}(t_x^0, x) = (x, \alpha(\mathrm{x}))$.

- Inductive case: Let us assume that $\texttt{value}_{\overline{\textsf{wr}}}(t_x^{i-1}, x) = \texttt{value}_{\overline{\textsf{wr}}}(t_x^0, x)$ and let us prove that $\texttt{value}_{\overline{\textsf{wr}}}(t_x^i, x) = (x, \alpha(\mathrm{x}))$. Note that in round i only keys associated to the variables of literals v_i^0, v_i^1 and v_i^2 are written; so for every other key x, $t_x^i = t_x^{i-1}$ and by induction hypothesis, $\texttt{value}_{\overline{\textsf{wr}}}(t_x^i, x) = (x, \alpha(\mathrm{x}))$. Let thus $j, 0 \leq j \leq 2$ s.t. $\alpha(v_i^j) = 1$. In this case, $t^i_{v_i^j} = t^i_{v_i^{j+2 \bmod 3}} = t_i^{j+2 \bmod 3}$ and $t^i_{v_i^{j+1 \bmod 3}} = t_i^{j+1 \bmod 3}$. Hence, we can conclude the inductive step as:

$$\begin{aligned}
\texttt{value}_{\overline{\textsf{wr}}}(t_i^{j+2 \bmod 3}, v_i^j) &= (v_i^j, 1) = (v_i^j, \alpha(v_i^j)) \\
\texttt{value}_{\overline{\textsf{wr}}}(t_i^{j+1 \bmod 3}, v_i^{j+1 \bmod 3}) &= (v_i^{j+1 \bmod 3}, 0) = (v_i^{j+1 \bmod 3}, \alpha(v_i^{j+1 \bmod 3})) \\
\texttt{value}_{\overline{\textsf{wr}}}(t_i^{j+2 \bmod 3}, v_i^{j+2 \bmod 3}) &= (v_i^{j+2 \bmod 3}, 0) = (v_i^{j+2 \bmod 3}, \alpha(v_i^{j+2 \bmod 3}))
\end{aligned}$$

We can thus conclude that $\overline{h}$ is a witness of h_φ. Let $i, j, 1 \leq i \leq n, 0 \leq j \leq 2$ be indices s.t. $\alpha(v_i^j) = 1$. For simplifying notation, we denote by t_0, t_1, t_2 to the transactions $t_i^j, t_i^{j+1 \bmod 3}$ and $t_i^{j+2 \bmod 3}$ respectively. We also denote by r_0, r_1, r_2 to the read events that belong to t_0, t_1 and t_2 respectively as well by v_0, v_1, v_2 to the keys associated to t_0, t_1 and t_2 respectively. For every key $x \neq v_0, v_1, v_2$ and for every transaction t that writes x, $\texttt{WHERE}(r_j)(\texttt{value}_{\textsf{wr}}(t, x)) = 0, 0 \leq j \leq 2$; so we can focus only on keys v_0, v_1 and v_2. Three cases arise:

- Transaction t_0: Let thus x be a key in $\{v_0, v_1, v_2\}$. By construction of h_φ and $\textsf{co}$, t_0 reads x from t_x^{i-1}. As proved before, $\texttt{value}_{\overline{\textsf{wr}}}(t_x^{i-1}, x) = (x, \alpha(x))$ and $\alpha(\mathrm{x}) = (x, 1)$ if and only if $x = v_0$. Hence, as $\texttt{WHERE}(r_0)(\texttt{value}_{\overline{\textsf{wr}}}(t_x^{i-1}, x)) = 0$ we conclude that $\texttt{WHERE}(r_0)(\texttt{value}_{\overline{\textsf{wr}}}(w_x^{r_0}, x)) = 0$.

- Transaction t_1: In this case, t_1 reads v_2 from $t_{v_2}^{i-1}$ and it reads v_0 and v_1 from t_0. On one hand, $\texttt{value}_{\overline{\textsf{wr}}}(t_{v_2}^{i-1}, v_2) = (v_2, \alpha(v_2)) = (v_2, 0)$. Thus, as $\texttt{WHERE}(r_1)(t_{v_2}^{i-1}) = 0$, we conclude that $\texttt{WHERE}(r_1)(\texttt{value}_{\overline{\textsf{wr}}}(w_{v_2}^{r_1}, v_2)) = 0$. On the other hand, by construction of h_φ, $\texttt{WHERE}(r_1)(\texttt{value}_{\overline{\textsf{wr}}}(t_0, v_0)) = \texttt{WHERE}(r_1)(\texttt{value}_{\overline{\textsf{wr}}}(t_0, v_1)) = 0$. Thus, the result hold.

- Transaction t_2: In this case, t_2 read v_0 from t_0 and v_1 and v_2 from t_1. By construction of h_φ both $\texttt{WHERE}(r_2)(\texttt{value}_{\overline{\textsf{wr}}}(t_0, v_0))$, $\texttt{WHERE}(r_2)(\texttt{value}_{\overline{\textsf{wr}}}(t_1, v_1))$ and $\texttt{WHERE}(r_2)(\texttt{value}_{\overline{\textsf{wr}}}(t_1, v_2))$ are equal to 0; so we conclude the result.

□

We conclude the proof showing that the execution $\xi = (\overline{h}, \textsf{co})$ is a consistent execution of h_φ. We observe that by construction of $\overline{\textsf{wr}}$ and $\textsf{co}$, $\overline{h}$ satisfies $\textsf{SER}$ using $\textsf{co}$. Corollary 4.4.17 proves that $\textsf{iso}(h_\varphi)$ is weaker than $\textsf{SER}$; which allow us to conclude that so $\overline{h}$ satisfies $\textsf{iso}(h_\varphi)$. In other words, that $\overline{h}$ is consistent.

Proposition 4.4.16. *Let $h = (T, \textsf{so}, \textsf{wr})$ be a full history, $\xi = (h, \textsf{co})$ be an execution of h, r be a* $\texttt{read}$ *event, t_2 be a transaction distinct from $\textsf{tr}(r)$, x be a key and $\textsf{v} \in \textsf{vis}(\textsf{iso}(h)(\textsf{tr}(e)))$. If $\textsf{v}(t_2, r, x)$ holds in ξ, then $(t_2, \textsf{tr}(r)) \in \textsf{co}$.*

Proof. The proposition is result of an immediate induction on the definition of $\textsf{v}$. The base case is $\textsf{po}, \textsf{so}, \textsf{wr} \subseteq \textsf{co}$, which holds by definition. The inductive case follows from the operators employed in Equation 2.3: union, composition and transitive closure of relations; which are monotonic. □

As a consequence of Proposition 4.4.16 and Serializability axiom definition, we obtain the following result.

Corollary 4.4.17. *Any isolation level is weaker than* SER*.*

Lemma 4.4.18. *If h_φ is consistent then φ is 1-in-3 satisfiable.*

Proof. If h_φ is consistent, there exists a consistent witness of h_φ $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$. As $\overline{h}$ is consistent and $\mathsf{iso}(h)$ is stronger than $\mathtt{RC}$, there exists a consistent execution of $\overline{h}$, $\xi = (\overline{h}, \mathsf{co})$. Let $\alpha_h : \mathtt{var}(\varphi) \to \{0, 1\}$ s.t. for every variable $v_j, 1 \leq j \leq m$, $\alpha_h(v_j) = 1$ if and only if $(0_j, 1_j) \in \mathsf{co}$. We show that φ is 1-in-3 satisfiable using α.

As an intermediate step, we prove that α_h describes the value read by any transaction in $\overline{h}$. For every $i, 0 \leq i \leq n$ and key x s.t. the variable x $\in \mathtt{var}(\varphi)$, let $t_x^i = \max_{\mathsf{co}}\{t \mid t \text{ writes } x \land \mathtt{round}(t) \leq i\}$. We prove by induction that for every $i, 0 \leq i \leq n$ (1) $\mathtt{value}_{\overline{\mathsf{wr}}}(t_x^i, x) = (x, \alpha(\mathrm{x}))$, (2) for any $\mathtt{read}$ event r from a transaction t s.t. $\mathtt{round}(t) \leq i$, if $(w, r) \in \overline{\mathsf{wr}}_x$, then w coincides with $\max_{\mathsf{co}}\{t \in T \mid t \text{ writes } x \land (t, \mathsf{tr}(r)) \in \mathsf{co}\}$ and (3) if $i > 0$, $\alpha(v_i^j) = 1$ if and only if $(t_i^j, t_i^{j+1 \bmod 3}) \in \mathsf{co}$ and $(t_i^{j+1 \bmod 3}, t_i^{j+2 \bmod 3}) \in \mathsf{co}$.

- Base case: Let $j, 1 \leq j \leq m$ be the integer s.t. $x = v_j$. In such case, (1) holds as $t_x^0 = 1_j$ if and only if $\alpha(v_j) = 1$; and in such case, $\mathtt{value}_{\overline{\mathsf{wr}}}(t_x^0, v_j) = (v_j, \alpha(v_j))$. Also (2) trivially holds as there is no $\mathtt{read}$ event in a transaction belonging to round 0. Finally, (3) also trivially holds as $i = 0$.

- Inductive case: We assume that (1), (2) and (3) hold for round $i - 1$ and let us prove it for round i. Let j the index of the co-minimal transaction among t_i^1, t_i^2, t_i^3. We denote by t_0, t_1, t_2 to $t_i^j, t_i^{j+1 \bmod 3}$ and $t_i^{j+2 \bmod 3}$ respectively, by r_0, r_1, r_2 to the unique $\mathtt{read}$ event in t_0, t_1 and t_2 respectively and by v_0, v_1 and v_2 to the keys associated to t_0, t_1 and t_2 respectively.

  Let thus $x \in \mathtt{var}(\varphi)$ be a key, t be a transaction among t_0, t_1, t_2 and let w_x^t be a transaction s.t. $(w_x^t, t) \in \overline{\mathsf{wr}}_x$. As $\mathtt{round}(t_x^{i-1}) < \mathtt{round}(t)$, $(t_x^{i-1}, t) \in \mathsf{wr}_{t_x^{i-1}}$. Hence, as $\overline{h}$ satisfies $\mathtt{RC}$, either $w_x^t = t_x^{i-1}$ or $\mathtt{round}(w_x^t) = i$.

  First we prove (3) analyzing t_0. As $(t_0, t_1) \in \mathsf{co}$ and $(t_0, t_2) \in \mathsf{co}$ and $\overline{\mathsf{wr}} \subseteq \mathsf{co}$ we deduce that $w_x^t = t_x^{i-1}$. In such case, as (1) holds by induction hypothesis and $\mathtt{WHERE}(r_0)(\mathtt{value}_{\overline{\mathsf{wr}}}(t_x^{i-1}, x)) = 0$, we conclude that $\alpha(x) = 1$ if $x = v_0$ and $\alpha(x) = 0$ if $x = v_1, v_2$.

  For proving (2) we analyze three cases depending on t:

  - $\underline{t = t_0}$: As proved before, if $t = t_0$, $w_x^t = t_x^{i-1}$. By definition of t_x^{i-1}, (2) holds.
  - $\underline{t = t_1}$: As t_0 only writes v_0 and v_1 and $(t_1, t_2) \in \mathsf{co}$ we deduce that for every key $x \neq v_0, v_1$, $w_x^{t_1} = t_x^{i-1}$; which immediately implies (2). As (3) holds for round i, we know that $\alpha(v_0) = 1$ and $\alpha(v_1) = 0$. Thus, if $x = v_0, v_1$, $\mathtt{WHERE}(r_2)(\mathtt{value}_{\overline{\mathsf{wr}}}(t_x^{i-1}, x)) = 1$; so $(t_x^{i-1}, t_1) \notin \overline{\mathsf{wr}}_x$. In conclusion, $w_x^{t_1} = t_0$; which implies (2) by definition of t_0.

- $\underline{t = t_2}$: As t_0, t_1 only write v_0, v_1 and v_2 we deduce that for every other key, $w_x^{t_2} = t_x^{i-1}$; which implies (2). Otherwise, we analyze the three sub-cases arising:
  - $\underline{x = v_2}$: In this case, t_0 does not write v_2; so there is only two options left, t_x^{i-1} and t_1. As (3) holds for round i, $\alpha(v_2) = 0$. Thus, as by induction hypothesis (2) holds for round $i-1$, $\texttt{value}_{\overline{\texttt{wr}}}(t_{v_2}^{i-1}, v_2) = (v_2, 0)$ and hence, $\texttt{WHERE}(r_2)(\texttt{value}_{\overline{\texttt{wr}}}(t_{v_2}^{i-1}, v_2)) = 1$. Therefore, $w_{v_2}^{t_2}$ must be t_1; which implies (2).
  - $\underline{x = v_0}$: Once again, there is only two possible options as t_1 does not write v_0. As (3) holds for round i, $\alpha(v_0) = 1$. Thus, as by induction hypothesis (2) holds for round $i-1$, $\texttt{value}_{\overline{\texttt{wr}}}(t_{v_0}^{i-1}, v_0) = (v_0, 1)$ and hence, $\texttt{WHERE}(r_2)(\texttt{value}_{\overline{\texttt{wr}}}(t_{v_0}^{i-1}, v_0)) = 1$. Therefore, $w_{v_0}^{t_2}$ must be t_0; which implies (2).
  - $\underline{x = v_1}$: We observe in this case that $\texttt{value}_{\overline{\texttt{wr}}}(t_0, v_1) = (x, 1)$; so $\texttt{WHERE}(r_2)(\texttt{value}_{\overline{\texttt{wr}}}(t_0, v_1)) = 1$. Therefore, there is only two possible options, t_1 and t_x^{i-1}. As $\overline{h}$ satisfies RC and $(t_1, t_2) \in \overline{\texttt{wr}}_{v_2}$, if $(t_x^{i-1}, t_2) \in \overline{\texttt{wr}}_{v_1}$, we deduce that $(t_1, t_x^{i-1}) \in \texttt{co}$. However, as $\texttt{round}(t_x^{i-1}) < \texttt{round}(t_1)$, $(t_x^{i-1}, t_1) \in \texttt{wr}_{t_x^{i-1}}$; which is impossible as $\overline{\texttt{wr}} \subseteq \texttt{co}$. Thus, we conclude that $w_{v_2}^{t_2} = t_1$; which implies (2).

For proving (1) we observe that for every key $x \neq v_0, v_1, v_2$, $t_x^i = t_x^{i-1}$ and by induction hypothesis we conclude that $\texttt{value}_{\overline{\texttt{wr}}}(t_x^i, x) = (x, \alpha(x))$. Moreover, as $(t_0, t_1) \in \texttt{co}$ and $(t_1, t_2) \in \texttt{co}$, $t_{v_0}^i = t_{v_2}^i = t_2$ and $t_{v_1}^i = t_1$. In addition, as (3) holds, $\alpha(v_0) = 1$ and $\alpha(v_1) = \alpha(v_2) = 0$. This allow us to conclude (1) also for the keys v_0, v_1 and v_2; so the inductive step is proven.

After proving (1), (2) and (3) we can conclude that φ is 1-in-3 satisfiable. For every round i, we observe that by (1) $\texttt{value}_{\overline{\texttt{wr}}}(t_x^i, x) = (x, \alpha(\text{x}))$. Moreover, as (2) holds, $(t_x^i, t_0^i) \in \overline{\texttt{wr}}_x$; where t_0^i is the first transaction in $\texttt{co}$ among the transactions in round i. As $\overline{h}$ is a witness of h_φ, $\texttt{WHERE}(r_0^i)(\texttt{value}_{\overline{\texttt{wr}}}(t_x^i, x)) = 0$; where r_0^i is the read event of t_0^i. Hence, exactly one variable among v_i^0, v_i^1 and v_i^2 has 1 as image by α. Therefore, φ is 1-in-3 satisfiable.

□

4.4.4 Checking Consistency of Partial Observation Histories

The proof of Theorem 4.4.8 relies on using non-trivial predicates in `WHERE` clauses. We also prove that checking consistency of client histories is NP-complete irrespectively of the complexity of these predicates. This result uses another class of histories, called *partial-observation* histories. These histories are a particular class of client histories where events read all inserted keys, irrespectively of their `WHERE` clauses (as if these clauses where *true*).

Definition 4.4.19. *A* partial observation history $h = (T, \texttt{so}, \texttt{wr})$ *is a client history for which there is a witness* $\overline{h} = (T, \texttt{so}, \overline{\texttt{wr}})$ *of* h*, s.t. for every* x*, if* $(w, r) \in \overline{\texttt{wr}}_x \setminus \texttt{wr}_x$*,* w *deletes* x*.*

Theorem 4.4.20. *Checking consistency of partial observation histories with bounded isolation configurations stronger than* RC *is NP-complete.*

The proof of NP-hardness uses a novel reduction from 3-SAT. The main difficulty for obtaining consistent witnesses of partial observation histories is the ambiguity of which delete event is responsible for each absent row.

The structure of the proof is divided in two parts: proving that the problem is NP and proving that it is NP-hard. The first part, corresponding to Lemma 4.4.21, is straightforward as, for any client history, we simply guess a suitable witness and a total order on its transactions for deducing its consistency applying Definition 4.3.1. The second part, corresponding to Lemmas 4.4.27 and 4.4.28 is more complicated. We use a novel reduction from 3-SAT. We encode a boolean formula φ in a history h_φ, s.t. h_φ is consistent iff φ is satisfiable.

We first prove that the problem is indeed in NP (Lemma 4.4.21).

Lemma 4.4.21. *The problem of checking consistency for a client history with an isolation configuration stronger than* $\mathtt{RC}$ *is in NP.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ a client history whose isolation configuration is stronger than $\mathtt{RC}$. Guessing a witness of h, $\overline{h}$, and an execution of $\overline{h}$, $\xi = (\overline{h}, \mathsf{co})$, can be done in $\mathcal{O}(|\mathsf{Keys}| \cdot |\mathsf{events}(h)|^2 + |T|^2) \subseteq \mathcal{O}(|h|^3)$. By Lemma 4.4.2, checking if ξ is consistent is equivalent as checking if $\textsc{saturate}(\overline{h}, \mathsf{co})$ is an acyclic relation. As by Lemma 4.4.4, Algorithm 5 requires polynomial time, we conclude the result.

□

For showing NP-hardness, we reduce 3-SAT to the problem of checking consistency of a partial observation history. Note that the problem is NP-hard in the case where the isolation configuration is not saturable, as discussed in Section 4.4.3, using the results in [29]. Therefore, we only prove it for the case where the isolation configuration is saturable.

Let $\varphi = \bigwedge_{i=1}^{n} C_i$ a CNF expression with at most 3 literals per clause (i.e. $C_i = l_i^1 \vee l_i^2 \vee l_i^3$). Without loss of generality we can assume that each clause contains exactly three literals and each literal in a clause refers to a different variable. We denote $\mathtt{var}(l_i^j)$ to the variable that the literal l_i^j employs and $\mathsf{Vars}(\varphi)$ the set of all variables of φ.

We design a history h_φ with an arbitrary saturable isolation configuration encoding φ. Thus, checking φ-satisfiability would reduce to checking h_φ's consistency. Note that as $\mathsf{iso}(h_\varphi)$ is saturable, h_φ's consistency is equivalent to checking pco's acyclicity; where $\mathsf{pco} = \textsc{saturate}(h_\varphi, (\mathsf{so} \cup \mathsf{wr})^+)$. We use the latter characterization of consistency for encoding the formula φ in h_φ.

First of all, we consider every literal in φ independently. This means that even if two literals l_i^j and $l_{i'}^{j'}$ share its variable ($\mathtt{var}(l_i^j) = \mathtt{var}(l_{i'}^{j'})$) we will reason independently about them. For that, we employ keys $\mathtt{var}(l_i^j)_i$ and $\mathtt{var}(l_{i'}^{j'})_{i'}$. We later enforce additional constraints for ensuring that $\mathtt{var}(l_i^j)_i$ and $\mathtt{var}(l_{i'}^{j'})_{i'}$ coordinate so assignments on $\mathtt{var}(l_i^j)_i$ coincide with assignments in $\mathtt{var}(l_{i'}^{j'})_{i'}$. For simplicity in the explanation, whenever we talk about a literal l that is negated (for example $l := \neg x$), we denote by $\neg l$ to the literal x. Also, we use indistinguishably x when referring to a variable in φ or to a homonymous key in h_φ. In addition, with the aim of simplifying the explanation, we assume hereinafter that any occurrence of indices i, i', j, j' satisfy that $1 \leq i, i' \leq n$ and $1 \leq j, j' \leq 3$.

$\texttt{INSERT}(\{\texttt{var}(l_i)^{+}_{(i,1)} : 1\})$
$\ldots$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{+}_{(i,i-1)} : 1\})$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{+}_{(i,i+1)} : 1\})$
$\ldots$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{+}_{(i,n)} : 1\})$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{-}_{(1,i)} : 1\})$
$\ldots$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{-}_{(i-1,i)} : 1\})$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{-}_{(i+1,i)} : 1\})$
$\ldots$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{-}_{(n,i)} : 1\})$
$\texttt{INSERT}(\{\texttt{var}(c_i^j) : 1\})$
$\texttt{DELETE}(\lambda r : \mathsf{key}(r) = \mathsf{var}(l_i^j)_i)$

(a) Transaction t_i^j

$\texttt{INSERT}(\{\texttt{var}(l_i)^{-}_{(i,1)} : 1\})$
$\ldots$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{-}_{(i,i-1)} : 1\})$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{-}_{(i,i+1)} : 1\})$
$\ldots$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{-}_{(i,n)} : 1\})$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{+}_{(1,i)} : 1\})$
$\ldots$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{+}_{(i-1,i)} : 1\})$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{+}_{(i+1,i)} : 1\})$
$\ldots$
$\texttt{INSERT}(\{\texttt{var}(l_i)^{+}_{(n,i)} : 1\})$
$\texttt{INSERT}(\{\texttt{var}(c_i^j) : 1\})$
$\texttt{INSERT}(\{\texttt{var}(c_i^{j-1 \bmod 3}) : 1\})$
$\texttt{DELETE}(\lambda r : \mathsf{key}(r) = \mathsf{var}(l_i^j)_i)$

(b) Transaction $\neg t_i^j$

$\texttt{INSERT}(\{x \in \texttt{A}_i^j : 0\})$
$\texttt{SELECT}(\lambda r : \mathsf{true})$

(c) Transaction S_i^j

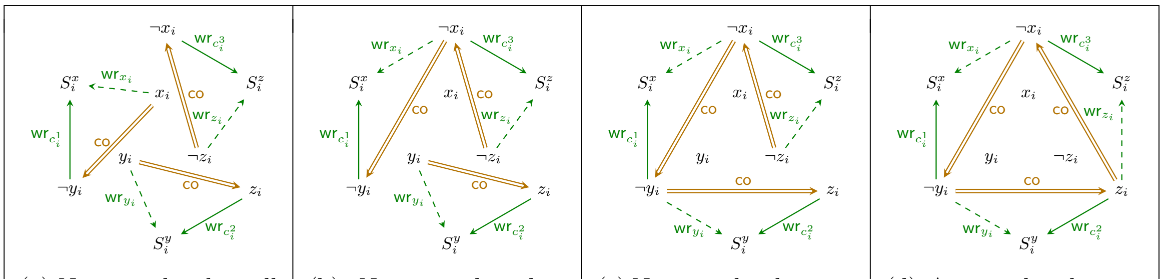
Figure 4.8: Description in full detail of the transactions t_i^j, $\neg t_i^j$ and S_i^j described in the proof of theorem 4.4.20 assuming $\texttt{sign}(t_i^j) = +$; where $\texttt{A}_i^j$ is the set of auxiliary variables for S_i^j. The case where $\texttt{sign}(t_i^j) = -$ is analogous replacing in the first two instructions of both t_i^j and $\neg t_i^j$ $+$ by $-$ and vice versa.

For every clause $C_i = l_i^1 \vee l_i^2 \vee l_i^3$, we create nine transactions denoted by t_i^j, $\neg t_i^j$ and S_i^j. Figure 4.8 shows in detail their definition, which we explain and justify during the following lines. The transaction t_i^j represents the state where l_i^j is satisfied while $\neg t_i^j$ represents the state where l_i^j is unsatisfied. Transaction S_i^j is in charge of selecting one of the two states. With this goal on mind, transactions t_i^j and $\neg t_i^j$ contain a `DELETE` event that deletes the key $\texttt{var}(l_i^j)_i$ while S_i^j contains a `SELECT` event that does *not* read $\texttt{var}(l_i^j)_i$ in h_φ. By Definition 4.4.19, any witnesses h' of h_φ must read $\texttt{var}(l_i^j)_i$ from a transaction that deletes it. As h_φ contain only two transactions that deletes such key (t_i^j and $\neg t_i^j$), we can interpret that if S_i^j reads $\texttt{var}(l_i^j)_i$ from t_i^j in h', then l_i^j is satisfied in φ while otherwise it is not.

For simplifying notation, as transactions $t_i^j, \neg t_i^j, S_i^j$ only have one read event, we define write-read dependencies directly from transactions instead of their read events. We also denote by $\texttt{var}(t_i^j)$ and $\texttt{var}(\neg t_i^j)$ to the variable $\texttt{var}(l_i^j)$, associating each transaction with the variable of its associated literal.

We divide the construction of the history h_φ in two main parts. In the first part, we relate transactions $t_i^j, \neg t_i^j$ and S_i^j with the clause C_i, ensuring a satisfying valuation of clause C_i corresponds to a consistent history when restricted to its associated transactions. In the second part, we link transactions associated to different clauses (for example t_i^j with $t_{i'}^{j'}$, $i \neq i'$), for ensuring that valuations are consistent between clauses (i.e. a variable is not assigned 1 in clause C_i and 0 in clause $C_{i'}$).

For the first part of h_φ's construction, we observe that "at least one literal among l_i^1, l_i^2 or l_i^3 must be satisfied" is equivalent to "$\neg l_i^1$, $\neg l_i^2$ and $\neg l_i^3$ cannot be satisfied at the same



(a) No co-cycle when all literals are satisfied.

(b) No co-cycle when two literals are satisfied

(c) No co-cycle when one literal is satisfied

(d) A co-cycle when no literal is satisfied

Figure 4.9: Transformation of the clause $C_i = x_i \vee y_i \vee \neg z_i$ into part of the history. Solid wr-edges in h_φ represent the constraints of the clause while dashed wr-edges, belonging only to the witnesses of h_φ, reflect the literals satisfied.

time". Thus, we add write-read dependencies to the history in such a way that if the three values that do not satisfy the clause are read by a witness h' of h_φ, axiom Read Committed forces h' to be inconsistent. We use an auxiliary key c_i^j written by transactions t_i^j, $\neg t_i^j$ and $\neg t_i^{(j+1) \bmod 3}$ and read by transaction S_i^j; enforcing $(\neg t_i^{(j+1) \bmod 3}, S_i^j) \in \mathsf{wr}_{c_i^j}$. Thanks to variable c_i^j, if $(\neg t_i^j, S_i^j) \in \mathsf{wr}_{\mathtt{var}(l_i^j)_i}$ in such witness h', for any consistent execution of h' with commit order co, $(\neg t_i^j, \neg t_i^{j+1 \bmod 3}) \in \mathsf{co}$. Hence, if h' is consistent, for every i there must exist a j s.t.$(t_i^j, S_i^j) \in \mathsf{wr}_{\mathtt{var}(l_i^j)_i}$. Otherwise, every commit order witnessing h''s consistency would be cyclic; which is a contradiction.

In Figure 4.9 we see how such co-cycle arise on any commit order witnessing h_φ's consistency; where φ contains the clause $C_i = x_i \vee y_i \vee \neg z_i$.

$$\mathtt{sign}(t) = \begin{cases} + & \text{if } t = t_i^j \wedge l_i^j = \mathtt{var}(l_i^j) \\ - & \text{if } t = t_i^j \wedge l_i^j = \neg\mathtt{var}(l_i^j) \\ - & \text{if } t = \neg t_i^j \wedge l_i^j = \mathtt{var}(l_i^j) \\ + & \text{if } t = \neg t_i^j \wedge l_i^j = \neg\mathtt{var}(l_i^j) \\ \bot & \text{otherwise} \end{cases} \quad \mathtt{opsign}(t) = \begin{cases} + & \text{if } \mathtt{sign}(t) = - \\ - & \text{if } \mathtt{sign}(t) = + \\ \bot & \text{otherwise} \end{cases} \tag{4.5}$$

For the second part of h_φ's construction, we utilize the functions `sign` and `opsign` described in Equation 4.5. The function `sign` describes when a literal l_i^j is *positive* (i.e. $l_i^j = \mathtt{var}(l_i^j)$) or *negative* (i.e. $l_i^j = \neg\mathtt{var}(l_i^j)$). If l_i^j is positive, it assigns to transaction t_i^j the symbol $+$ and to $\neg t_i^j$ the symbol $-$; while if l_i^j is negative, the opposite. Such symbol is denoted the *sign* of a transaction. Hence, for each transaction t_i s.t. $\mathtt{sign}(t_i) \neq \bot$ (i.e. t_i is either t_i^j or $\neg t_i^j$), we introduce $n-1$ `INSERT` events, one per key $\mathtt{var}(l_i^j)_{(i,i')}^{\mathtt{sign}(t_i)}$, $i' \neq i$, that write on that exact key.

The auxiliary keys $\mathtt{var}(l_i^j)_{(i,i')}^{\mathtt{sign}(t_i)}$ and $\mathtt{var}(l_i^j)_{(i,i')}^{\mathtt{opsign}(t_i)}$ are key to ensure consistency between clauses. Intuitively, if S_i^j reads $\mathtt{var}(l_i^j)_i$ from a positive transaction t in a consistent execution of h_φ, $\xi = (\overline{h}, \mathsf{co})$, and t' is a negative transaction s.t. $\mathtt{var}(t) = \mathtt{var}(t')$, then

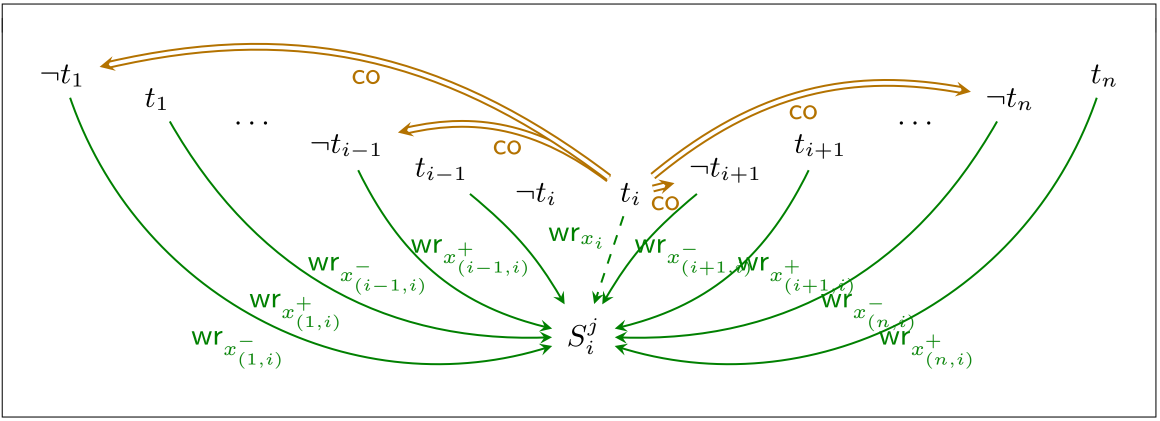

$\neg t_1$
t_1
$\dots$
$\neg t_{i-1}$
t_{i-1}
$\neg t_i$
t_i
$\neg t_{i+1}$
t_{i+1}
$\dots$
$\neg t_n$
t_n
co
co
co
co
wr_{x_i}
$\mathsf{wr}_{x^-_{(1,i)}}$
$\mathsf{wr}_{x^+_{(1,i)}}$
$\mathsf{wr}_{x^-_{(i-1,i)}}$
$\mathsf{wr}_{x^+_{(i-1,i)}}$
$\mathsf{wr}_{x^-_{(i+1,i)}}$
$\mathsf{wr}_{x^+_{(i+1,i)}}$
$\mathsf{wr}_{x^-_{(n,i)}}$
$\mathsf{wr}_{x^+_{(n,i)}}$
S^j_i

Figure 4.10: Commit edges between transactions of different sign associated to variable $x = \mathtt{var}(l^j_i)$. Superindices j are omitted for legibility. For simplicity on the Figure, we assume that $\mathtt{sign}(t_k) = +$ and $\mathtt{sign}(\neg t_k) = -$; the situation generalizes for any other setting. If S^j_i would read x_i from t_i in a witness h' of h_φ (respectively $\neg t_i$), for every $i' \neq i$ $(t_i, \neg t_{i'}) \in \mathsf{co}$, (resp. $(\neg t_i, t_{i'}) \in \mathsf{co}$).

$(t, t') \in \mathsf{co}$; where $\overline{h}$ is a witness of h_φ. Hence, any other transaction $S^{j'}_{i'}$ must read $\mathtt{var}(l^{j'}_{i'})_{i'}$ also from a positive transaction in $\overline{h}$; otherwise co would be cyclic, which is impossible as co must be a total order. This phenomenon, that is depicted in Figure 4.10, ensures that $\mathtt{var}(l^j_i)$ is always read from transactions with the same sign. In conclusion, we can establish consistent valuation of the variables of φ based on the write-read dependencies of the witnesses of h_φ.

We introduce a succint final part on the construction of h_φ for technical reasons. Indeed, any witness of h_φ ensures that wr_x^{-1} is a total function for any $x \in \mathsf{Keys}$. We impose a few additional constraints on h_φ so we can better characterize the witnesses of h_φ. First, we assume that there exists an initial transaction that inserts, for every key x, a dummy value different from $\dagger_x$ (for example 0). Then, we impose that t^j_i and $\neg t^j_i$ read every key x from the initial transaction. Finally, for the case of transactions S^j_i, we define the set of *auxiliary keys* $\mathtt{A}^j_i$ that contain every key different from $c^j_i, \mathtt{var}(l^j_i)_i, \mathtt{var}(l^j_i)^+_{(i',i)}$ and $\mathtt{var}(l^j_i)^-_{(i',i)}$. We introduce on S^j_i an `INSERT` event that writes every key in $\mathtt{A}^j_i$ with an abritrary value (for example, 0). Hence, S^j_i reads every key in A^j_i from its own insert and no extra write-read dependency is required.

With this technical addendum, we define $h_\varphi = (T, \mathsf{so}, \mathsf{wr})$ as the conjunction of all transactions and relations described above. In such case, the only information missing in h_φ to be a full history is $\mathsf{wr}^{-1}_{\mathtt{var}(l^j_i)_i}(S^j_i)$. We assume that no more variables than the ones aforementioned belong to Keys.

The proof of NP-hardness goes as follows: first, we prove in Lemma 4.4.22 that h_φ is indeed a polynomial transformation of φ. Then, as $\mathsf{iso}(h_\varphi)$ is saturable, by Theorem 4.4.5 we observe that it suffices to prove that φ is satisfiable if and only there is a witness $\overline{h}$ of h_φ s.t. the relation $\mathsf{pco}_{\overline{h}} = \textsc{saturate}(\overline{h}, (\mathsf{so} \cup \overline{\mathsf{wr}})^+)$ is acyclic. For simplifying the reasoning when

$\mathsf{iso}(h_\varphi)$ has an arbitrary isolation configuration, we rely on Lemma 4.4.23 for reducing the proof at the case when $\mathsf{iso}(h_\varphi) = \mathtt{RC}$.

Hence, we prove on Lemma 4.4.27 that on one hand, whenever φ is satisfiable we can construct a witness $\overline{h}$ of h_φ based on such assignment for which $\mathsf{pco}_{\overline{h}}$ is acyclic. For that, we require Lemmas 4.4.24 to 4.4.26.

On the other hand, whenever there is a consistent witness $\overline{h}$ of h_φ, we prove on Lemma 4.4.28 that we can construct a satisfying assignment of φ based on the write-read dependencies in $\overline{h}$. In this case, we require once more Lemma 4.4.24.

Lemma 4.4.22. *The history h_φ has polynomial size on the length of φ.*

Proof. Let φ a CNF with n clauses and 3 literals per clause. Then, as φ has $3n$ literals, h_φ employs $9n$ transactions plus one additional one ($\mathtt{init}$). The number of keys, $|\mathsf{Keys}|$, is quadratic on n as transactions t_i^j and $\neg t_i^j$ insert $\mathcal{O}(n)$ keys while S_i^j only insert keys also inserted by other transactions. Moreover, the number of events in h_φ, $\mathsf{events}(h_\varphi)$ is in $\mathcal{O}(|\mathsf{Keys}|) = \mathcal{O}(n^2)$ as transactions $t_i^j, \neg t_i^j$ have one $\mathtt{INSERT}$ event per keys inserted (and they insert $\mathcal{O}(n)$ keys) and one $\mathtt{DELETE}$ event and transactions S_i^j only have two events. In addition, $\mathsf{so} \subseteq T \times T$ and $\mathsf{wr} \subseteq \mathsf{Keys} \times \mathsf{events}(h_\varphi) \times \mathsf{events}(h_\varphi)$; so their size is also polynomial on n. Thus, h_φ is a polynomial transformation of φ. □

One caveat of h_φ is that its isolation configuration is unknown. Lemma 4.4.23 express that, in the particular case of h_φ, all saturable isolation levels stronger than $\mathtt{RC}$ are equivalent (they impose the same constraints). Hence, thereinafter we can assume without loss of generality that $\mathsf{iso}(h_\varphi) = \mathtt{RC}$.

Lemma 4.4.23. *Under history h_φ, $\mathsf{iso}(h_\varphi)$ is equivalent to $\mathtt{RC}$ (i.e. $\mathsf{iso}(h_\varphi)$ is both weaker and stronger than $\mathtt{RC}$).*

Proof. Let $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ be any witness of h_φ and let $h^{\mathtt{RC}}$ be the history that only differ with $\overline{h}$ on its isolation configuration ($\mathsf{iso}(h^{\mathtt{RC}}) = \mathtt{RC}$ instead of $\mathsf{iso}(h)$). We prove that $\overline{h}$ and $h^{\mathtt{RC}}$ are simultaneously consistent or inconsistent.

As both $\mathsf{iso}(h_\varphi)$ and $\mathtt{RC}$ are saturable, by Theorem 4.4.5, the proof is equivalent to prove that $\mathsf{pco}_{\overline{h}}$ and $\mathsf{pco}_{\mathtt{RC}}$ are simultaneously cyclic or acyclic; where $\mathsf{pco}_{\overline{h}} = \textsc{saturate}(\overline{h}, (\mathsf{so} \cup \overline{\mathsf{wr}})^+)$ and $\mathsf{pco}_{\mathtt{RC}} = \textsc{saturate}(h^{\mathtt{RC}}, (\mathsf{so} \cup \overline{\mathsf{wr}})^+)$. We prove that the two relations coincide, which allow us to conclude the result.

We observe that as $\mathsf{iso}(\overline{h})$ is weaker than $\mathtt{RC}$, $\mathsf{pco}_{\mathtt{RC}} \subseteq \mathsf{pco}_{\overline{h}}$. Thus, it suffices to prove that $\mathsf{pco}_{\overline{h}} \subseteq \mathsf{pco}_{\mathtt{RC}}$. Let t, t' be two transactions s.t. $(t, t') \in \mathsf{pco}_{\overline{h}}$ and let us prove that $(t, t') \in \mathsf{pco}_{\mathtt{RC}}$. As $(\mathsf{so} \cup \overline{\mathsf{wr}})^+ \subseteq \mathsf{pco}_{\mathtt{RC}}$; we can assume without loss of generality that $(t, t') \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$. In such case, there exists $r \in \mathsf{reads}(\overline{h}), x \in \mathsf{Keys}$ and $v \in \mathsf{vis}(\mathsf{iso}(\overline{h})(\mathsf{tr}(r)))$ s.t. $t' = \overline{\mathsf{wr}}_x{}^{-1}(r)$ and $\mathsf{v}(\mathsf{pco}_{\overline{h}})(t, r, x)$ holds in $\overline{h}$. As $\mathsf{iso}(\overline{h})$ is saturable, $(t, r) \in (\mathsf{so} \cup \overline{\mathsf{wr}})^+$.

First, we note that $\mathsf{tr}(r) \neq \mathtt{init}$ as it does not contain any read event. As t' is a $(\mathsf{so} \cup \overline{\mathsf{wr}})^+$-predecessor of $\mathsf{tr}(r)$, and transactions S_i^j are $(\mathsf{so} \cup \overline{\mathsf{wr}})$-maximal, t' is not a S_i^j transaction; so it must be a t_i^j transaction. However, note that by construction of transactions t_i^j, their only $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessor is $\mathtt{init}$. Thus, their only $(\mathsf{so} \cup \overline{\mathsf{wr}})$-succesors can be transactions $S_{i'}^{j'}$; transactions that do not have $(\mathsf{so} \cup \overline{\mathsf{wr}})$-successors. In conclusion, if $(t', \mathsf{tr}(r)) \in (\mathsf{so} \cup \overline{\mathsf{wr}})^+$, $(t', \mathsf{tr}(r)) \in \mathsf{so} \cup \overline{\mathsf{wr}}$, and therefore, $(t', r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$. □

Lemma 4.4.24 states a characterization of all commit order cycles imposed by the axiom $\mathtt{RC}$ that only relate the nine transactions associated to a clause in φ.

Lemma 4.4.24. *Let $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ a witness of h_φ. For a fixed i, there is a $\mathsf{pco}_{\overline{h}}$-cycle relating $\mathtt{init}$, $t_i^j, \neg t_i^j$ and S_i^j, $1 \leq j \leq 3$ in h if and only if for all $1 \leq j \leq 3$, $(\neg t_i^j, S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)}$ in $\overline{h}$.*

Proof. A graphical description of the different cases of this proof can be seen in Figure 4.9.

$\underline{\Longleftarrow}$

Let us suppose that for every j, $1 \leq j \leq 3$, $(\neg t_i^j, S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)_i}$. As $\neg t_i^j$ writes c_i^j and $(\neg t_i^{(j+1) \bmod 3}, S_i^j) \in \overline{\mathsf{wr}}_{c_i^j}$, by axiom $\mathtt{RC}$ we deduce that, $(\neg t_i^j, \neg t_i^{(j+1) \bmod 3}) \in \mathsf{pco}_{\overline{h}}$. Therefore, there is a $\mathsf{pco}_{\overline{h}}$-cycle between transactions $\neg t_i^1, \neg t_i^2$ and $\neg t_i^3$.

$\underline{\Longrightarrow}$

First, note that $\mathsf{so} \cup \overline{\mathsf{wr}}$ is acyclic, so any $\mathsf{pco}_{\overline{h}}$-cycle has to include at least one $\mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$-dependency. Hence, let t, t' be distinct transactions such that $(t', t) \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$ is an edge belonging to such cycle. By axiom $\mathsf{Read\ Committed}$, this implies that there exists a read event r and a key x s.t. $(t, r) \in \overline{\mathsf{wr}}_x$ and $(t, r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$. Note that in particular this means that t' and t are two distinct $(\mathsf{so} \cup \overline{\mathsf{wr}})$-succesors of $\mathsf{tr}(r)$.

We observe that $\mathsf{tr}(r) \neq \mathtt{init}$ as $\mathtt{init}$ does not contain any read event. Moreover, $\mathsf{tr}(r) \neq t_i^j, \neg t_i^j$ as those transactions have only one $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessor, $\mathtt{init}$. Hence, there exists j s.t. $\mathsf{tr}(r) = S_i^j$. In this case, every ke written by t_i^j or $\neg t_i^j$ besides c_i^j and $\mathtt{var}(l_i^j)_i$ is read by S_i^j from the $\mathtt{INSERT}$ event in its own transaction. We distinguish between two cases:

- $\underline{x = \mathtt{var}(l_i^j)_i}$: The only transactions that write $\mathtt{var}(l_i^j)_i$ are t_i^j, $\neg t_i^j$, $\mathtt{init}$ and transactions $S_{i'}^{j'}$. However, transactions $S_{i'}^{j'}$ have only one $(\mathsf{so} \cup \overline{\mathsf{wr}})$-succesor, $\mathtt{init}$ in h_φ. As $\forall x \neq \mathtt{var}(l_i^j)_i, \overline{\mathsf{wr}}_x{}^{-1}(S_i^j) = \mathsf{wr}_x{}^{-1}(S_i^j)$, one of them, $\mathtt{init}$ must be either t or t'. However, $t \neq \mathtt{init}$ as $\mathtt{init}$ does not delete $\mathtt{var}(l_i^j)_i$; so $t = \mathtt{init}$. But in such case, $(t', t) \in \mathsf{so}$; which contradicts that $(t', t) \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$. This proves that this case is impossible.

- $\underline{x = c_i^j}$: In such case, as $(\neg t_i^{j+1 \bmod 3}, S_i^j) \in \mathsf{wr}_{c_i^j}$, $t = \neg t_i^{j+1 \bmod 3}$. The only transactions that writes c_i^j and are $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessors of S_i^j are $\mathtt{init}, t_i^j$ and $\neg t_i^j$. As $(\mathtt{init}, t) \in \mathsf{so}$; $t \neq \mathtt{init}$. Thus, any of the other two transactions are candidates to be t'. Note that $(t', t) \in \mathsf{pco}_{\overline{h}}$ is part of a cycle; so let t'' be a transaction s.t. $(t'', t') \in \mathsf{pco}_{\overline{h}}$.

  If $(t'', t') \in (\mathsf{so} \cup \overline{\mathsf{wr}})$ would hold, as for every key x, $\overline{\mathsf{wr}}_x{}^{-1}(t) = \mathsf{wr}_x{}^{-1}(t)$, $t'' = \mathtt{init}$. As (t', t) is part of a $\mathsf{pco}_{\overline{h}}$ cycle and $t \neq \mathtt{init}$, there must exist a transaction $t''' \neq t'' = \mathtt{init}$ s.t. $(t''', t'') \in \mathsf{pco}_{\overline{h}}$ is part of such cycle. Note that $(t''', \mathtt{init}) \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})$ by construction of h_φ. Hence, there exists a key y and a read event r' s.t. $(\mathtt{init}, r') \in \overline{\mathsf{wr}}_y$ and $(t''', r') \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$. By construction of h_φ, if $(\mathtt{init}, r') \in \overline{\mathsf{wr}}_y$ then $\mathsf{tr}(r')$ must be $t_i^{j'}$ for some j'. But as we mentioned earlier, such transactions only have one $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessor, $\mathtt{init}$; so it is impossible that $(t'', t') \in (\mathsf{so} \cup \overline{\mathsf{wr}})$.

  Hence, $(t'', t') \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})$. Replicating the same argument as before we can deduce that there exists a j' s.t. $(t'', S_i^{j'}) \in \overline{\mathsf{wr}}_{c_i^{j'}}$, $t' = \neg t_i^{j'+1 \bmod 3}$ and t'' is either $t_i^{j'}$ or

$\neg t_i^{j'}$. However, as discussed before, t' could only be $\neg t_i^j$ or t_i^j. Therefore, $t' = \neg t_i^j$ and $j = j' + 1 \bmod 3$.

Finally, as $t'' \neq t$, there must exist a transaction t''' s.t. $(t''', t'') \in \mathsf{pco}_{\overline{h}}$. By the same argument once more, there exists an index j'' s.t. $t'' = \neg t_i^{j''+1 \bmod 3}$, $(t''', S_i^{j''}) \in \overline{\mathsf{wr}}_{c_i^{j''}}$ and t''' is either $t_i^{j''}$ or $\neg t_i^{j''}$. Once more, as t'' could only be $\neg t_i^{j'}$ or $t_i^{j'}$, we deduce that $j' = j'' + 1 \bmod 3$ and $t'' = \neg t_i^{j'}$. Note that in this case $j = j'' + 2 \bmod 3$. Thus, $t = \neg t_i^{j+1 \bmod 3} = \neg t_i^{j''} = t'''$. In conclusion, if such cycle exists it contain exactly the transactions $\neg t_i^1, \neg t_i^2$ and $\neg t_i^3$ and for each of them, $(\neg t_i^j, S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)_i}$.

□

Lemma 4.4.25 states that any $\mathsf{pco}_{\overline{h}}$-dependencies imposed by the axiom $\mathtt{RC}$ on transactions t, t' associated to diferent clauses in φ are related to valuation choices of literals in φ.

Lemma 4.4.25. *Let $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ a witness of the history h_φ. For every pair of transactions t, t' and indices i, j, if $\mathtt{var}(t) = \mathtt{var}(t')$, t' deletes $\mathtt{var}(l_i^j)_i$, $t \neq \neg t_i^{j+1 \bmod 3}$ and $(t', t) \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$ in h, then $(t', S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)_i}$.*

Proof. Let i, j be indices and t, t' be distinct transactions such that $t \neq \neg t_i^{j+1 \bmod 3}$ and $(t', t) \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$. Hence, $(t', t) \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$, by axiom Read Committed, there must exist a key x and a read event r s.t. $(t, r) \in \overline{\mathsf{wr}}_x$, t' writes x and $(t', r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$. We characterize the possible candidates of transactions $t, t', \mathtt{tr}(r)$ and key x.

First, as $\mathtt{tr}(r)$ has two different $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessors, $\mathtt{tr}(r) \neq \mathtt{init}, t_{i'}^{j'}$; for any indices i', j'. Hence, there must exist indices i', j' s.t. $\mathtt{tr}(r) = S_{i'}^{j'}$.

Next we deduce that t and t' belongs to different clauses. As t' deletes $\mathtt{var}(l_i^j)_i$, we deduce that t' is either t_i^j or $\neg t_i^j$. Hence, as $t \neq \neg t_i^{j+1 \bmod 3}$, neither t_i^j nor $\neg t_i^j$ are $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessors of S_i^j but both t and t' are $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessors of S_i^j, we then deduce that t and t' belong to different clauses.

Finally, we deduce that $i' = i$ and $(t', S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)_i}$. As x is written by t and t' and $x \notin \mathtt{A}_{i'}^{j'}$; either t or t' are associated to the same clause as $S_{i'}^{j'}$. If t would be associated to clause $C_{i'}$, then t should be either $t_{i'}^{j'}$ or $\neg t_{i'}^{j'}$ and $x = \mathtt{var}(l_{i'}^{j'})_{i'}$. However, this contradicts that t' writes $\mathtt{var}(l_{i'}^{j'})_{i'}$ as t' is either t_i^j or $\neg t_i^j$. Hence, as t is not associated to clause $C_{i'}$, $i' = i$. As $(t', S_i^j) \notin (\mathsf{so} \cup \mathsf{wr})$ but $(t', S_i^j) \in (\mathsf{so} \cup \overline{\mathsf{wr}})$ and $\overline{\mathsf{wr}}_y = \mathsf{wr}_y$ for any key $y \neq \mathtt{var}(l_i^j)_i$, we conclude that $(t', S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)_i}$. □

Lemma 4.4.26 states that $\mathsf{pco}_{\overline{h}}$ does not contain tuples of transactions associated to literals with equal variable and sign.

Lemma 4.4.26. *Let $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ a witness of the history h_φ. For every pair of transactions t, t' and indices i, j, if $\mathtt{sign}(t) = \mathtt{sign}(t')$, $\mathtt{var}(t) = \mathtt{var}(t') = \mathtt{var}(l_i^j)$, $(t, S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)_i}$ and $t' \neq \neg t_i^{j-1 \bmod 3}$ then $(t', t) \notin \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$.*

Proof. We reason by contradiction. Let us suppose that t, t' are a pair of transactions such that $\texttt{sign}(t) = \texttt{sign}(t')$, $\texttt{var}(t) = \texttt{var}(t') = \texttt{var}(l_i^j)$, $(t, S_i^j) \in \overline{\mathsf{wr}}_{\texttt{var}(t)_i}$, $t' \neq \neg t^{j-1 \bmod 3}$ and $(t', t) \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$, for some indices i, j. As $(t, S_i^j) \in \overline{\mathsf{wr}}_{\texttt{var}(l_i^j)_i}$, t is either t_i^j or $\neg t_i^j$. Moreover, as $(t', t) \in \mathsf{pco}_{\overline{h}} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$, by axiom Read Committed we deduce that there exists a key x and a read event r s.t. $(t', r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$, $(t, r) \in \overline{\mathsf{wr}}_x$ and t' writes x.

We first prove that t' and t are associated to different clauses. As $(\texttt{init}, t) \in \mathsf{so}$, $t' \neq \texttt{init}$. Next, as $(t', r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$ and transactions $S_{i'}^{j'}$ are $\mathsf{so} \cup \overline{\mathsf{wr}}$-maximal, we deduce that there must exist a pair of indices i', j' s.t. $t' = t_{i'}^{j'}$ or $\neg t_{i'}^{j'}$. Moreover, as $t' \neq \neg t_i^{j-1 \bmod 3}$, t is either t_i^j or $\neg t_i^j$. In addition, as in any witness of h_φ both t_i^j and $\neg t_i^j$ cannot be $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessors of S_i^j, we deduce that $i' \neq i$.

Finally we contradict the hypothesis proving that $\texttt{sign}(t) \neq \texttt{sign}(t')$. If $i' \neq i$, $t \neq \texttt{init}$ but t is a $\overline{\mathsf{wr}}$-predecessor of $\mathsf{tr}(r)$, there must exist indices i'', j'' s.t. $\mathsf{tr}(r) = S_{i''}^{j''}$. Hence, as $x \in \mathtt{A}_{i''}^{j''}$ and it is written by t and t', i'' must be either i' or i. However, $i'' \neq i$ as in that case, $x = \texttt{var}(l_i^j)_i$ and t' does not write $\texttt{var}(l_i^j)_i$. Hence, $i'' = i' \neq i$ and $x = \texttt{var}(l_i^j)_{(i',i)}^{\texttt{sign}(t')}$. However, as t writes x, by construction of h_φ, we must conclude that $\texttt{sign}(t) \neq \texttt{sign}(t')$. Thus, as we reached a contradiction, the lemma holds. □

Lemma 4.4.27. *For every boolean formula φ, if φ is satisfiable then there is a consistent witness $\overline{h}$ of h_φ.*

Proof. Let $\alpha : \mathsf{Vars}(\varphi) \to \{0, 1\}$ an assignment that satisfies φ. Let $h_\varphi^\alpha = (T, \mathsf{so}, \overline{\mathsf{wr}})$ the extension of h_φ s.t. for every i, j, $(t_i^j, S_i^j) \in \overline{\mathsf{wr}}_{\texttt{var}(l_i^j)_i}$ if $l_i^j[\alpha(\texttt{var}(l_i^j))/\texttt{var}(l_i^j)] = \mathsf{true}$ and $(\neg t_i^j, S_i^j) \in \overline{\mathsf{wr}}_{\texttt{var}(l_i^j)_i}$ otherwise. Note that for every two transactions t, t' s.t. $\texttt{var}(t) = \texttt{var}(t')$, $\alpha(\texttt{var}(t)) = \alpha(\texttt{var}(t'))$. Hence, if $(t, S_i^j) \in \overline{\mathsf{wr}}_{\texttt{var}(t)_i}$ and $(t', S_{i'}^{j'}) \in \overline{\mathsf{wr}}_{\texttt{var}(t')_{i'}}$ then $\texttt{sign}(t) = \texttt{sign}(t')$. In addition, by construction of h_φ, for every transaction S_i^j, the only variable x such that $\mathsf{wr}_x^{-1}(S_i^j)\uparrow$ is $x = \texttt{var}(l_i^j)$. Thus, for every $x \in \mathsf{Keys}$, $\overline{\mathsf{wr}}_x{}^{-1}$ is defined for any read that does not read locally and therefore, h_φ^α is a full history that extends h_φ.

Let us prove that h_φ^α is consistent. As mentioned before, thanks to Theorem 4.4.5, we can reduce the problem of checking if h_φ^α is consistent to the problem of checking if $\mathsf{pco}_{h_\varphi^\alpha} = \textsc{saturate}(h_\varphi^\alpha, (\mathsf{so} \cup \overline{\mathsf{wr}})^+)$ is acyclic.

We reason by contradiction, assuming there is a $\mathsf{pco}_{h_\varphi^\alpha}$-cycle and reaching a contradiction. Clearly $\mathsf{so} \cup \overline{\mathsf{wr}}$ is acyclic as $\mathsf{so} \cup \mathsf{wr}$ is acyclic, transactions S_i^j are $(\mathsf{so} \cup \overline{\mathsf{wr}})$-maximal and $\overline{\mathsf{wr}} \setminus \mathsf{wr}$ only contains tuples (t_i^j, S_i^j) or $(\neg t_i^j, S_i^j)$. Thus, any $\mathsf{pco}_{h_\varphi^\alpha}$-cycle in h_φ^α contains at least one edge $(t', t) \in \mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$; so let be t, t' such a pair of distinct transactions s.t. $(t', t) \in \mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$ and (t', t) is part of the $\mathsf{pco}_{h_\varphi^\alpha}$-cycle.

First, we observe that by construction of h_φ, transactions S_i^j are $(\mathsf{so} \cup \overline{\mathsf{wr}})$-maximal. Moreover, they also $\mathsf{pco}_{h_\varphi^\alpha}$-maximal: by contradiction, if there was a transaction u_i^j s.t. $(S_i^j, u_i^j) \in \mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})+$, by axiom Read Committed, there would be a variable a read event r s.t. $(S_i^j, r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$; which is impossible. We also observe that $\texttt{init}$ is not only $(\mathsf{so} \cup \overline{\mathsf{wr}})$-minimal but $\mathsf{pco}_{h_\varphi^\alpha}$-minimal. By the same argument, if there would be a transaction

$u \neq \texttt{init}$ s.t. $(u, \texttt{init}) \in \mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$, by axiom Read Committed, there should be a read event r and a key x s.t. $(\texttt{init}, r) \in \overline{\mathsf{wr}}_x$ and $(u', r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$. However, by construction of h_φ, the only transactions that read a variable from $\texttt{init}$ are t_i^j; transactions with only one $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessor. This shows that such transaction u does not exist. Altogether, the $\mathsf{pco}_{h_\varphi^\alpha}$-cycle can only contain pairs of transactions t_i^j and $\neg t_i^j$. In particular, as transactions t_i^j have only one $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessor, $\texttt{init}$, such $\mathsf{pco}_{h_\varphi^\alpha}$-cycle is in $\mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$.

Next, we note that, as every clause C_i is satisfied by α, there exists an index j s.t. $(t_i^j, S_i^j) \in \texttt{var}(l_i^j)$. By Lemma 4.4.24, we know there is no $\mathsf{pco}_{h_\varphi^\alpha}$-cycle relating the nine transactions associated with clause C_i and $\texttt{init}$. Therefore, a $\mathsf{pco}_{h_\varphi^\alpha}$-cycle has to involve at least two transactions from different clauses. Hence, we can assume without loss of generality that t and t' belong to the same clause.

As $(t', t) \in \mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$, there must exist a key x and a read event r_x s.t. t writes x, $(t, r_x) \in \overline{\mathsf{wr}}_x$ and $(t', r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$. By construction of h_φ, the only case when two transactions from different clauses write the same variable is when $\texttt{var}(t) = \texttt{var}(t')$. In particular, as t and t' belong to different clauses, there must exist indices i, i' s.t. $x = \texttt{var}(t)^{\texttt{sign}(t)}_{(i', i)}$ and $\mathsf{tr}(r_x) = S_i^j$. Hence, there is only one candidate for transaction t': t_i^j if $\texttt{sign}(t_i^j) = \texttt{sign}(t') = \texttt{opsign}(t)$ and $\neg t_i^j$ otherwise. Therefore, as $t', \mathsf{tr}(r_x)$ belong to the same clause and t' is a $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessor of $\mathsf{tr}(r_x)$, we conclude that $(t', S_i^j) \in \overline{\mathsf{wr}}_{\texttt{var}(t')_i}$.

To reach a contradiction, we find a pair of distinct transactions $\tilde{t}, \hat{t}$ in the $\mathsf{pco}_{h_\varphi^\alpha}$-cycle from different clauses but associated to the same variable. First, as (t', t) is part of the $\mathsf{pco}_{h_\varphi^\alpha}$-cycle, there exists a $\mathsf{pco}_{h_\varphi^\alpha}$-predecessor of t', t'' s.t. $(t'', t') \in \mathsf{pco}_{h_\varphi^\alpha}$ is part of the $\mathsf{pco}_{h_\varphi^\alpha}$-cycle. As we mentioned before, $(t'', t') \in \mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$. Then, there must exist a key y and a read event r_y s.t. t'' writes y, $(t', r_y) \in \overline{\mathsf{wr}}_y$ and $(t'', r) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$. Two cases arise:

- <u>t'' is not associated to clause i</u>: In this case, as both t', t'' write variable y, by construction of h_φ we observe that $\texttt{var}(t'') = \texttt{var}(t')$. Thus, we denote $\tilde{t} = t''$ and $\hat{t} = t'$.

- <u>t'' is associated to clause i</u>: In this case, $t'' \neq \neg t'$ as no transaction in h have both t' and $\neg t'$ as $(\mathsf{so} \cup \overline{\mathsf{wr}})$-predecessors. Hence, as no clause has two literals referring to the same variable, $\texttt{var}(t') \neq \texttt{var}(t'')$. Thus, as t'' and t' have one common key, we deduce that $t'' = \neg t_i^{j-1 \bmod 3}$ and $y = c_i^{j-1 \bmod 3}$. Thus, as $(t', r) \in \overline{\mathsf{wr}}_{c_i^{j-1 \bmod 3}}$, we can conclude that $\mathsf{tr}(r_y) = S_i^{j-1 \bmod 3}$ and $(t'', S_i^{j-1 \bmod 3}) \in \overline{\mathsf{wr}}_{\texttt{var}(t'')_i}$. As $t'' \neq t$, there must exist a transaction t''' s.t. $(t''', t'') \in \mathsf{pco}_{h_\varphi^\alpha}$ belongs to the $\mathsf{pco}_{h_\varphi^\alpha}$-cycle. Again, we observe two cases:

  - <u>t''' is not associated to clause i</u>: In this case, by an analogous argument, we observe that $\texttt{var}(t''') = \texttt{var}(t'')$. Thus, we denote $\tilde{t} = t'''$ and $\hat{t} = t''$.

  - <u>t''' is associated to clause i</u>: By the same reasoning as before, $t''' = \neg t_i^{j-2 \bmod 3}$ and $(t''', S_i^{j-2 \bmod 3}) \in \overline{\mathsf{wr}}_{\texttt{var}(t''')_i}$. Moreover, as $t''' \neq t$, there must exist a transaction t'''' s.t. $(t'''', t''') \in \mathsf{pco}_{h_\varphi^\alpha}$ belongs to the $\mathsf{pco}_{h_\varphi^\alpha}$-cycle. Moreover, t'''' is not associated to clause i, as, once more, we would deduce that $t'''' = \neg t_i^{j-3 \bmod 3}$ and that $(t'''', S_i^{j-3 \bmod 3}) \in \overline{\mathsf{wr}}_{\texttt{var}(t'''')_i}$; which is impossible as by the construction of h_φ^α is satisfied. Hence, t'''' and t''' belong to different clauses and $\mathtt{var}(t'''') = \mathtt{var}(t''')$. We denote in this case $\tilde{t} = t''''$ and $\hat{t} = t'''$.

Finally, we reach a contradiction with the help of Lemmas 4.4.26 and 4.4.25. On one hand, by the choice of transactions $\hat{t}$ and $\tilde{t}$, we know that $\mathtt{var}(\hat{t}) = \mathtt{var}(\tilde{t})$ and there exist indices $\tilde{i}, \tilde{j}$ s.t. $\tilde{t}$ $\mathsf{deletes}$ $\mathtt{var}(l_{\tilde{i}}^{\tilde{j}})$. Moreover, $\hat{t} \neq \neg t_{\tilde{i}}^{\tilde{j}+1 \text{ mod } 3}$ as they belong to different clauses. Thus, as $(\tilde{t}, \hat{t}) \in \mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$, by Lemma 4.4.25 we deduce that $(\tilde{t}, S_{\tilde{i}}^{\tilde{j}}) \in \overline{\mathsf{wr}}_{\mathtt{var}(\tilde{t})_i}$. On the other hand, we also know that there exist indices $\hat{i}, \hat{j}$ s.t. $\hat{t}$ is associated to the literal $l_{\hat{i}}^{\hat{j}}$ and $(\hat{t}, S_{\hat{i}}^{\hat{j}}) \in \overline{\mathsf{wr}}_{\mathtt{var}(t)_{\hat{i}}}$. Hence, by construction of h_φ^α, as $\mathtt{var}(\hat{t}) = \mathtt{var}(\tilde{t})$, $(\tilde{t}, S_{\tilde{i}}^{\tilde{j}}) \in \overline{\mathsf{wr}}_{\mathtt{var}(\tilde{t})_i}$ and $(\hat{t}, _{\hat{i}}^{\hat{j}}) \in \overline{\mathsf{wr}}_{\mathtt{var}(\hat{t})_{\hat{i}}}$, we deduce that $\mathtt{sign}(\hat{t}) = \mathtt{sign}(\tilde{t})$. However, by Lemma 4.4.26, we deduce that $(\tilde{t}, \hat{t}) \notin \mathsf{pco}_{h_\varphi^\alpha} \setminus (\mathsf{so} \cup \overline{\mathsf{wr}})^+$. This contradicts that $(\tilde{t}, \hat{t}) h_\varphi^\alpha$ is part of the $\mathsf{pco}_{h_\varphi^\alpha}$-cycle. Thus, the initial hypothesis, that $\mathsf{pco}_{h_\varphi^\alpha}$ is cyclic, is false. In conclusion, $\mathsf{pco}_{h_\varphi^\alpha}$ is acyclic, so h_φ is consistent as h_φ^α is a consistent witness of h_φ.

□

Lemma 4.4.28. *For every boolean formula φ, if there is a consistent witness of h, then φ is satisfiable.*

Proof. Let $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ be a consistent witness of h_φ. Hence, by Theorem 4.4.5, the relation $\mathsf{pco}_{\overline{h}} = \textsc{saturate}(\overline{h}, (\mathsf{so} \cup \overline{\mathsf{wr}})^+)$ is acyclic. We use this fact to construct a satisfying assignment of φ. Let us call u_i^j to the transaction s.t. $(u_i^j, S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)_i}$. Note that by construction of h_φ, u_i^j $\mathsf{deletes}$ $\mathtt{var}(l_i^j)_i$, so u_i^j is either t_i^j or $\neg t_i^j$.

We first prove that for every pair of pairs of indices i, i', j, j', if $\mathtt{var}(u_i^j) = \mathtt{var}(u_{i'}^{j'})$ then $\mathtt{sign}(u_i^j) = \mathtt{sign}(u_{i'}^{j'})$. By contradiction, let $u_i^j, u_{i'}^{j'}$ be a pair of transactions s.t. $\mathtt{var}(u_i^j) = \mathtt{var}(u_{i'}^{j'})$ and $\mathtt{sign}(u_i^j) \neq \mathtt{sign}(u_{i'}^{j'})$. In such case, $\mathtt{opsign}(u_i^j) = \mathtt{sign}(u_{i'}^{j'})$. Thus, both transactions write $\mathtt{var}(u_i^j)_{(i',i)}^{\mathtt{opsign}(u_i^j)}$ and $\mathtt{var}(u_i^j)_{(i,i')}^{\mathtt{sign}(u_i^j)}$. By axiom $\mathsf{Read\ Committed}$, as $(u_{i'}^{j'}, S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(u_i^j)_{(i',i)}^{\mathtt{opsign}(u_i^j)}}$ and $(u_i^j, S_i^j) \in \overline{\mathsf{wr}}$, we conclude that $(u_i^j, u_{i'}^{j'}) \in \mathsf{pco}_{\overline{h}}$. By a symmetric argument using $\mathtt{var}(u_i^j)_{(i,i')}^{\mathtt{sign}(u_i^j)}$ we deduce that $(u_{i'}^{j'}, u_i^j) \in \mathsf{pco}_{\overline{h}}$. However, this is impossible as $\mathsf{pco}_{\overline{h}}$ is acylclic; so we conclude that indeed $\mathtt{sign}(u_i^j) = \mathtt{sign}(u_{i'}^{j'})$.

Next, we construct a map that assign at each variable in φ a value 0 or 1. Let $\alpha_h : \mathsf{Vars}(\varphi) \to \{0, 1\}$ be the map that assigns for each variable $\mathtt{var}(l_i^j)$ the value 1 if $\mathtt{sign}(u_i^j) = +$ and 0 if $\mathtt{sign}(u_i^j) = -$. Note that this map is well defined as, by the previous paragraph, if two literals $l_i^j, l_{i'}^{j'}$ share variable, then their respective transactions $u_i^j, u_{i'}^{j'}$ have the same sign.

Finally, we prove that φ is satisfied with this assignment. By construction of α_h, for every pair of indices i, j, $l_i^j[\alpha_h(\mathtt{var}(l_i^j))/\mathtt{var}(l_i^j)]$ is true if and only if $(t_i^j, S_i^j) \in \overline{\mathsf{wr}}_{\mathtt{var}(l_i^j)}$. Moreover, as $\mathsf{pco}_{\overline{h}}$ is acyclic, by Lemma 4.4.24, we know that for each i there exists a j s.t. $u_i^j \neq \neg t_i^j$. Hence, for this j, u_i^j must be t_i^j as u_i^j is either t_i^j or $\neg t_i^j$. Therefore, every clause is satisfied using α_h as assignment; so φ is satisfiable.

□

4.5 Effectively Checking Consistency of Client Histories

The result of Theorem 4.4.8 implicitly asks whether there exist conditions on the histories for which checking consistency remains polynomial as in [29]. We describe an algorithm for checking consistency of client histories and identify cases in which it runs in polynomial time.

4.5.1 An Algorithm for Checking Consistency of Client Histories

Consider a client history $h = (T, \mathsf{so}, \mathsf{wr})$ which is consistent. For every consistent witness $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ of h there exists a consistent execution of $\overline{h}$, $\xi = (\overline{h}, \mathsf{co})$. The commit order co contains $(\mathsf{so} \cup \mathsf{wr})^+$ and any other ordering constraint derived from axioms by observing that $(\mathsf{so} \cup \mathsf{wr})^+ \subseteq \mathsf{co}$. More generally, co includes all constraints generated by the least fixpoint of the function SATURATE defined in Algorithm 5 when starting from $(\mathsf{so}{\cup}\mathsf{wr})^+$ as partial commit order. This least fixpoint exists because SATURATE is monotonic. It is computed as usual by iterating SATURATE until the output does not change. We use $\mathsf{FIX}(\lambda R : \textsc{saturate}(h, R))(\mathsf{so} \cup \mathsf{wr})^+$ to denote this least fixpoint. In general, such a fixpoint computation is just an under-approximation of co, and it is not enough for determining h's consistency.

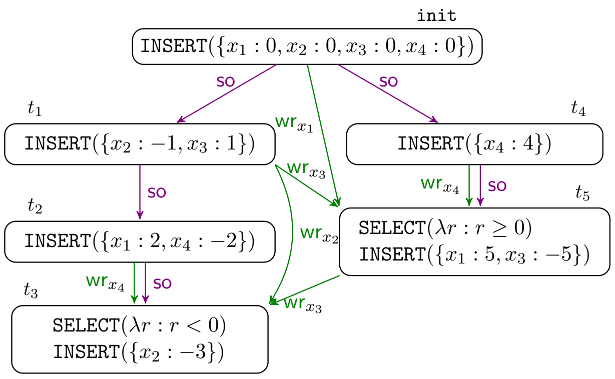
Algorithm 7 Checking consistency of client histories

```
1:  function checkConsistency($h = (T, \mathsf{so}, \mathsf{wr})$)
2:    let $\mathsf{pco} = \mathsf{FIX}(\lambda R : \textsc{saturate}(h, R))(\mathsf{so} \cup \mathsf{wr})^+$
3:    let $E_h = \{(r, x) \mid r \in \mathsf{reads}(h), x \in \mathsf{Keys}.\mathsf{wr}_x^{-1}(r) \uparrow \text{ and } \mathtt{1}_r^x(\mathsf{pco}) \neq \emptyset\}$
4:    let $X_h$ = the set of mappings that map each $(r, x) \in E_h$ to a member of $\mathtt{0}_x^r(\mathsf{pco})$
5:    if $\mathsf{pco}$ is cyclic then return false
6:    else if there exists $(r, x) \in E_h$ such that $\mathtt{0}_x^r(\mathsf{pco}) = \emptyset$ then return false
7:    else if $E_h = \emptyset$ then return exploreConsistentPrefixes($h, \emptyset$)
8:    else
9:      for all $f \in X_h$ do
10:       $\mathtt{seen} \leftarrow \emptyset$; $h' \leftarrow h \bigoplus_{(r,x) \in E_h} \mathsf{wr}_x(f(r, x), r)$
11:       if exploreConsistentPrefixes($h', \emptyset$) then return true
12:     return false
```

The algorithm we propose, described in Algorithm 7, exploits the partial commit order pco obtained by such a fixpoint computation (line 2) for determining h's consistency. For a read r, key x, we define $\mathtt{1}_x^r(\mathsf{pco})$, resp., $\mathtt{0}_x^r(\mathsf{pco})$, to be the set of transactions that are *not* committed after $\mathsf{tr}(r)$ and which write a value that satisfies, resp., does not satisfy, the predicate $\mathtt{WHERE}(r)$. The formal description of both sets can be seen in Equation 4.6.

$$\begin{aligned}
\mathtt{1}_x^r(\mathsf{pco}) &= \{t \in T \mid (\mathsf{tr}(r), t) \notin \mathsf{pco} \ \wedge\ \mathtt{WHERE}(r)(\mathtt{value}_{\mathsf{wr}}(t, x)) = 1\} \\
\mathtt{0}_x^r(\mathsf{pco}) &= \{t \in T \mid (\mathsf{tr}(r), t) \notin \mathsf{pco} \ \wedge\ \mathtt{WHERE}(r)(\mathtt{value}_{\mathsf{wr}}(t, x)) = 0\}
\end{aligned} \tag{4.6}$$

The set $\mathtt{0}_x^r(\mathsf{pco})$ can be used to identify extensions that are not witness of a history. Let us consider the client history h depicted in Figure 4.11a. Observe that t_3 is not reading x_1 and t_5 is not reading x_2. Table 4.11b describes all possible full extensions $\overline{h}$ of h. An


init
INSERT({$x_1 : 0, x_2 : 0, x_3 : 0, x_4 : 0$})
so
so
t_1
INSERT({$x_2 : -1, x_3 : 1$})
t_4
INSERT({$x_4 : 4$})
wr$_{x_1}$
wr$_{x_3}$
wr$_{x_4}$
so
so
t_2
INSERT({$x_1 : 2, x_4 : -2$})
wr$_{x_2}$
t_5
SELECT($\lambda r : r \geq 0$)
INSERT({$x_1 : 5, x_3 : -5$})
wr$_{x_4}$
so
t_3
wr$_{x_3}$
SELECT($\lambda r : r < 0$)
INSERT({$x_2 : -3$})

(a) A history where t_3, t_5 have PC and SER as isolation levels respectively. The isolation levels of the other transactions are unspecified.

History	$\mathsf{wr}_{x_1}^{-1}(t_3)$	$\mathsf{wr}_{x_2}^{-1}(t_5)$
h_1	init	init
h_2	init	t_1
h_3	init	t_3
h_4	t_2	init
h_5	t_2	t_1
h_6	t_2	t_3
h_7	t_5	init
h_8	t_5	t_1
h_9	t_5	t_3

(b) Table describing all possible full extensions of the history in Figure 4.11a.

History	$\mathsf{wr}_{x_1}^{-1}(t_3)$	$\mathsf{wr}_{x_2}^{-1}(t_5)$
h_{258}	**undef**	t_1

(c) Table describing the only conflict-free extension of Figure 4.11a.

Figure 4.11: Comparison between conflict-free extensions and full extensions of the history h in Figure 4.11a. In h, wr^{-1} is not defined for two pairs: (t_3, x_1) and (t_5, x_2); where we identify the single SELECT event in a transaction with its transaction. Table 4.11b describes all possible full extensions of h. For example, the first extension, h_1, states that $(\mathtt{init}, t_3) \in \mathsf{wr}_{x_1}$ and $(\mathtt{init}, t_5) \in \mathsf{wr}_{x_2}$. Algorithm 7 only explore the only extension h_{258} described in Table 4.11c; where $\mathsf{wr}_{x_1}^{-1}(t_3) \uparrow$ and $(t_1, t_5) \in \mathsf{wr}_{x_2}$. The history h_{258} can be extended to histories h_2, h_5 and h_8.

execution $\xi = (\overline{h}, \mathsf{co})$ is consistent if $(t, r) \in \overline{\mathsf{wr}}_x \setminus \mathsf{wr}_x$ implies $\mathtt{WHERE}(r)(\mathtt{value}_{\mathsf{wr}}(t, x)) = 0$. This implies that extensions h_1, h_4, and h_7, where $(\mathtt{init}, t_5) \in \overline{\mathsf{wr}}_{x_2}$, are not witnesses of h as $\mathtt{WHERE}(t_5)(\mathtt{value}_{\mathsf{wr}}(\mathtt{init}, x_2)) = 1$. We note that $\mathtt{init} \notin 0_{x_2}^{t_5}(\mathsf{pco}) = \{t_1\}$. Also, observe that $(t_5, t_3) \in \mathsf{wr}$; so extensions h_3, h_6 and h_9, where $(t_3, t_5) \in \overline{\mathsf{wr}}_{x_2}$, are not a witness of h. Once again, $t_3 \notin 0_{x_2}^{t_5}(\mathsf{pco})$. In general, for every read event r and key x s.t. $\mathsf{wr}_x^{-1}(r) \uparrow$, the extension of h where $(t, r) \in \overline{\mathsf{wr}}_x$, $t \notin 0_x^r(\mathsf{pco})$, is not a witness of h. In particular, if $\mathsf{wr}_x^{-1}(r) \uparrow$ but $0_x^r(\mathsf{pco}) = \emptyset$, then no witness of h can exist.

The sets $0_x^r(\mathsf{pco})$ are not sufficient to determine if a witness is a consistent witness as our previous example shows: $0_{x_1}^{t_3}(\mathsf{pco}) = \{\mathtt{init}, t_2, t_5\}$, but h_2 is not consistent.

Algorithm 7, combines an enumeration of history extensions with a search for a consistent execution of each extension. The extensions are *not* necessarily full.

In case $\mathsf{wr}_x^{-1}(r)$ is undefined, we use sets $1_x^r(\mathsf{pco})$ to decide whether the extension of h requires specifying $\mathsf{wr}_x^{-1}(r)$ for determining h's consistency. Algorithm 7 specifies $\mathsf{wr}_x^{-1}(r)$ only if (r, x) is a so-called *conflict*, i.e., $\mathsf{wr}_x^{-1}(r)$ is undefined and $1_x^r(\mathsf{pco}) \neq \emptyset$.

Following the example of Figure 4.11, we observe that $1_{x_1}^{t_3}(\mathsf{pco}) = \emptyset$, all transactions that write on x_1 write non-negative values; but instead $1_{x_2}^{t_5}(\mathsf{pco}) = \{\mathtt{init}\}$. Intuitively, this means that if some extension h' that does not specify $\mathsf{wr}_{x_1}^{-1}(t_3)$ does not violate any axiom when using some commit order co, then we can extend h', defining $\mathsf{wr}_{x_1}^{-1}(t_3)$ as some adequate transaction, and obtain a full history $\overline{h}$ s.t. the execution $\xi = (\overline{h}, \mathsf{co})$ is consistent. On the other hand, specifying the write-read dependency of t_5 on x_2 matters. For not contradicting any axiom using co, we may require $(\mathtt{init}, t_5) \in \overline{\mathsf{wr}}_{x_2}$. However, such extension is not even a witness of h as $\mathtt{WHERE}(\mathtt{init})(\mathtt{value}_{\mathsf{wr}}(\mathtt{init}, x_2)) = 1$. This intuition holds for the isolation levels that Algorithm 7 considers.

A history is *conflict-free* if it does not have conflicts. Our previous discussion reduces the problem of checking consistency of a history to checking consistency of its conflict-free extensions. For example, the history h in Figure 4.11a is not conflict-free but the extension h_{258} defined in Table 4.11c is. Instead of checking consistency of the nine possible extensions, we only check consistency of h_{258}.

Algorithm 7 starts by checking if there is at least a conflict-free extension of h (line 6). If h is conflict-free, it directly calls Algorithm 8 (line 7); while otherwise, it iterates over conflict-free extensions of h, calling Algorithm 8 on each of them (line 11).

Algorithm 8 check consistency of conflict-free histories

```
1: function EXPLORECONSISTENTPREFIXES($h = (T, \mathsf{so}, \mathsf{wr}), P$)
2:     if $|P| = |T|$ then return true
3:     for all $t \in T \setminus P$ s.t. $P \triangleright_t (P \cup \{t\})$ do
4:         if $\exists P' \in \mathsf{seen}$ s.t. $P' \equiv_{\mathsf{iso}(h)} (P \cup \{t\})$ then continue
5:         else if EXPLORECONSISTENTPREFIXES($h, P \cup \{t\}$) then return true
6:         else $\mathsf{seen} \leftarrow \mathsf{seen} \cup (P \cup \{t\})$
7:     return false
```

Algorithm 8 describes the search for the commit order of a conflict-free history h. This is a recursive enumeration of consistent prefixes of histories that backtracks when detecting inconsistency (it generalizes Algorithm 2 in [29]). A *prefix* of a history $h = (T, \mathsf{so}, \mathsf{wr})$ is a tuple $P = (T_P, M_P)$ where $T_P \subseteq T$ is a set of transactions and $M_P : \mathsf{Keys} \to T_P$ is a mapping s.t. (1) so predecessors of transactions in T_P are also in T_P, i.e., $\forall t \in T_P.\ \mathsf{so}^{-1}(t) \in T_P$ and (2) for every x, $M_P(x)$ is a so-maximal transaction in T_P that writes x (M_P records a last write for every key).

For every prefix $P = (T_P, M_P)$ of a history h and a transaction $t \in T \setminus T_P$, we say a prefix $P' = (T_{P'}, M_{P'})$ of h is an *extension* of P *using* t if $T_{P'} = T_P \cup \{t\}$ and for every key x, $M_{P'}(x)$ is t or $M_P(x)$. Algorithm 8 extensions, denoted as $P \cup \{t\}$, guarantee that for every key x, if t writes x, then $M_{P'}(x) = t$.

Extending the prefix P using t means that any transaction $t' \in T_P$ is committed before t. Algorithm 8 focuses on special extensions that lead to commit orders of consistent executions.

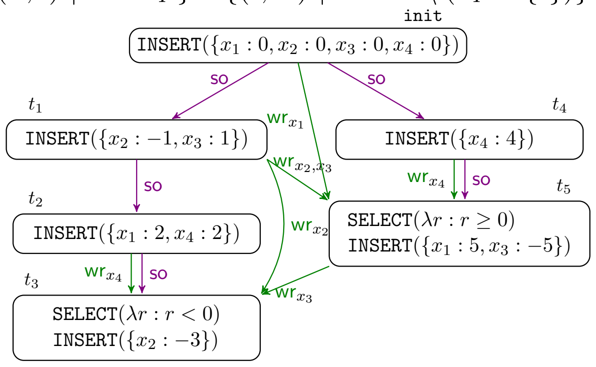
Definition 4.5.1. *Let h be a history, $P = (T_P, M_P)$ be a prefix of h, t a transaction that is not in T_P and $P' = (T_{P'}, M_{P'})$ be an exetension of P using t. The prefix P' is a* consistent extension *of P with t, denoted by $P \triangleright_t P'$, if*

1. *P is pco-closed: for every transaction $t' \in T_P$ s.t. $(t', t) \in \mathsf{pco}$ then $t' \in T_P$,*

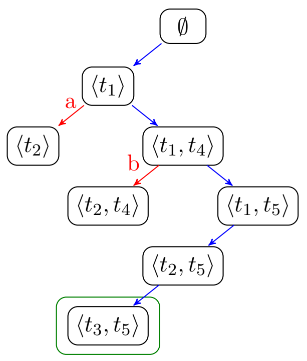
2. *t does not overwrite other transactions in P: for every* `read` *event r outside of the prefix,*

Axiom	**Predicate**
Serializability, Prefix, Read Atomic, Read Committed	$\nexists x \in \mathsf{Keys}$ s.t. t writes x, $\mathsf{wr}_x^{-1}(r)\downarrow$ $v(\mathsf{pco}_t^P)(t, r, x)$ holds in h and $\mathsf{wr}_x^{-1}(r) \in T_P$
Conflict	$\nexists x \in \mathsf{Keys}, t' \in T_P \cup \{t\}$ s.t. t' writes x, $\mathsf{wr}_x^{-1}(r)\downarrow$ $v(\mathsf{pco}_t^P)(t', r, x)$ holds in h and $\mathsf{wr}_x^{-1}(r) \neq M_P(x)$

Table 4.1: Predicates relating prefixes and visibility relations where pco_t^P is defined as $\mathsf{pco} \cup \{(t', t) \mid t' \in T_P\} \cup \{(t, t'') \mid t'' \in T \setminus (T_P \cup \{t\})\}$.



(a) Conflict-free history corresponding to the extension h_{258} (Table 4.11c) of the history in Figure 4.11a


$\emptyset$
$\langle t_1 \rangle$
a
$\langle t_2 \rangle$
$\langle t_1, t_4 \rangle$
b
$\langle t_2, t_4 \rangle$
$\langle t_1, t_5 \rangle$
$\langle t_2, t_5 \rangle$
$\langle t_3, t_5 \rangle$

(b) Execution of Algorithm 7 on the history in Figure 4.12a.

Figure 4.12: Applying Algorithm 8 on the conflict-free consistent history h_{258} on the left. The right part pictures a search for valid extensions of consistent prefixes on h_{258}. Prefixes are represented by their so-maximal transactions, e.g., $\langle t_2 \rangle$ contains all transactions which are before t_2 in so, i.e., $\{\mathtt{init}, t_1, t_2\}$. A red arrow means that the search is blocked (the prefix at the target is not a consistent extension), while a blue arrow mean that the search continues.

> *i.e.,* $\mathsf{tr}(r) \in T \setminus T_{P'}$ *and every visibility relation* $v \in \mathsf{vis}(\mathsf{iso}(h))(\mathsf{tr}(r))$*, the predicate* $\mathsf{vp}_v^P(t, r)$ *defined in Table 4.1 holds in* h*.*

We say that a prefix is consistent if it is either the empty prefix or it is a consistent extension of a consistent prefix.

Figure 4.12b depicts the execution of Algorithm 8 on the conflict-free history Figure 4.12a (history h_{258} from Table 4.11c). Blocked and effectuated calls are represented by read and blue arrows respectively. The read arrow a is due to condition 1 in Definition 4.5.1: as t_3 enforces $\mathtt{PC}$, reads x_4 from t_2, and t_4 is visible to it ($\mathsf{vis}_{\mathsf{Prefix}}(t_4, t_3, x_4)$), $(t_4, t_2) \in \mathsf{pco}$; so consistent prefixes can not contain t_2 if they do not contain t_4. The read arrow b is due to condition 2: as t_5 enforces $\mathtt{SER}$ and it reads x_4 from t_4, consistent prefixes can not contain t_2 unless t_5 is included. When reaching prefix $\langle t_3, t_5 \rangle$, the search terminates and deduces that h is consistent. From the commit order induced by the search tree we can construct the extension of h where missing write-read dependencies are obtained by applying the axioms on such a commit order. In our case, from $\mathtt{init} <_{\mathsf{co}} t_1 <_{\mathsf{co}} t_4 <_{\mathsf{co}} t_5 <_{\mathsf{co}} t_2 <_{\mathsf{co}} t_3$, we deduce that the execution $\xi = (h_5, \mathsf{co})$ is a consistent execution of h_{258}, and hence of h; where h_5 is the history described in Table 4.11b.

For complexity optimizations (see Section 4.5.2), Algorithm 8 requires an isolation level-dependent equivalence relation between consistent prefixes. If there is transaction $t \in T$ s.t. $\mathsf{iso}(h)(t) = \mathtt{SI}$, prefixes $P = (T_P, M_P)$ and $P' = (T_{P'}, M_{P'})$ are *equivalent* iff they are equal (i.e. $T_P = T_{P'}, M_P = M_{P'}$). Otherwise, they are *equivalent* iff $T_P = T_{P'}$.

Theorem 4.5.2. *Let h be a client history whose isolation configuration is defined using $\{\mathtt{SER}, \mathtt{SI}, \mathtt{PC}, \mathtt{RA}, \mathtt{RC}\}$. Algorithm 7 returns* $\mathtt{true}$ *if and only if h is consistent.*

The proof of Theorem 4.5.2 is a consequence of Lemmas 4.5.4 and 4.5.7.

Lemma 4.5.3. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history, $P = (T_P, M_P)$ be a consistent prefix of h and $t \in T \setminus T_P$. If $(P \cup \{t\}) \in \mathtt{seen}$ then* EXPLORECONSISTENTPREFIXES$(h, P \cup \{t\})$ *returns* false.

Proof. If $(P \cup \{t\}) \in \mathtt{seen}$, then $P \cup \{t\}$ has been to $\mathtt{seen}$ added at line 6 of Algorithm 8. To execute such instruction, the condition at line 4, EXPLORECONSISTENTPREFIXES$(h, P \cup \{t\})$ returns $\mathtt{true}$, does not hold; which let us conclude the result. □

Lemma 4.5.4. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history whose isolation configuration is stronger than* $\mathtt{RC}$. *If h is consistent, Algorithm 7 returns* $\mathtt{true}$.

Proof. Let h be a consistent history that satisfies the hypothesis of the Lemma. As h is consistent, let $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ be a witness of h and let $\xi = (\overline{h}, \mathsf{co})$ be a consistent execution of $\overline{h}$. We first reduce the problem to prove that Algorithm 8 returns true on a particular witness of h, a history $\hat{h}$ s.t. $h \subseteq \hat{h} \subseteq \overline{h}$.

First, let pco, E_h and X_h be defined as in Algorithm 7 at lines 2-4. As $\overline{h}$ is consistent, for every $\mathtt{read}$ event r and a variable x s.t. $\mathsf{wr}_x^{-1}(r)\uparrow$, $\overline{\mathsf{wr}}_x^{-1}(r)\downarrow$ and $\mathtt{WHERE}(r)(\mathtt{value}_{\mathsf{wr}}(t_x^r, x)) = 0$; where $t_x^r = \overline{\mathsf{wr}}_x^{-1}(r)$.

On one hand, if E_h is empty, X_h is empty as well. In such case, we denote $\hat{h} = h$. On the other hand, if $E_h \neq \emptyset$, for every $(r, x) \in E_h$ we know that the transaction t_x^r belongs to $\mathsf{0}_x^r$. Therefore, $X_h \neq \emptyset$. Thus, let f be the map that assigns for every pair $(r, x) \in E_h$ the transaction t_x^r; and let $\hat{h} = (T, \mathsf{so}, \hat{\mathsf{wr}})$ be the history s.t. $\hat{h} = h \bigoplus_{(r,x) \in E_h} \mathsf{wr}_x(f(r, x), r)$. We observe that the fact that $\hat{h}$ is a history and $\overline{h}$ witnesses $\hat{h}$'s consistency using co is immediate as $\mathsf{wr} \subseteq \hat{\mathsf{wr}} \subseteq \overline{\mathsf{wr}}$. Note that in both cases, the condition at line 6 does not hold. Therefore, to prove that Algorithm 7 returns $\mathtt{true}$ it suffices to prove that EXPLORECONSISTENTPREFIXES$(\hat{h}, \emptyset)$ returns $\mathtt{true}$.

We define an inductive sequence of prefixes based on co and show that they represent recursive calls to Algorithm 8. As a base case, let P_0 be the prefix with only $\mathtt{init}$ as transaction. Assuming that for every $j, 0 \leq j \leq i$, P_i is defined, let $P_{i+1} = P_i \cup \{t_i\}$; where t_i is the i-th transaction of T according to co. By construction of co, $\mathsf{pco} \subseteq \mathsf{co}$. Hence, Property 1 immediately holds. Moreover, as co witnesses $\overline{h}$'s consistency, Property 2 also holds; so $P_i \triangleright_{t_{i+1}} P_{i+1}$.

We conclude showing by induction on the number of transactions that are not in the prefix that for every $i, 0 \leq i \leq |T|$, EXPLORECONSISTENTPREFIXES$(\hat{h}, P_i)$ returns $\mathtt{true}$.

- Base case: The base case is $i = |T|$. In such case, $P_{|T|}$ contains all transactions in T. Therefore, the condition at line 2 in Algorithm 8 holds and the algorithm returns $\mathtt{true}$.

- Inductive case: The inductive hypothesis guarantees that for every $k, i \leq k \leq |T|$, exploreConsistentPrefixes$(\hat{h}, P_i)$ returns $\mathtt{true}$ and we show that exploreConsistentPrefixes$(\hat{h}, P_{i-1})$ also returns $\mathtt{true}$. By definition of P_i, $T_{P_i} = T_{P_{i-1}} \cup \{t_i\}$. In particular, $|P_i| \neq |T|$ and $P_{i-1} \triangleright_{t_{i+1}} P_i$. In addition, by induction hypothesis, we know that exploreConsistentPrefixes$(\hat{h}, P_i)$ returns $\mathtt{true}$. Hence, by Lemma 4.5.3, $P_i \not\in \mathtt{seen}$. Altogether, we deduce that exploreConsistentPrefixes$(\hat{h}, P_{i-1})$ returns $\mathtt{true}$.

□

Lemma 4.5.5. *Let $\hat{h} = (T, \mathsf{so}, \widehat{\mathsf{wr}})$ be a client history and $P = (T_P, M_P)$ be a consistent prefix. If* exploreConsistentPrefixes$(\hat{h}, P)$ *returns* $\mathtt{true}$*, there exist distinct transactions $t_i \in T\ T_P$ and a collection of consistent prefixes $P_i = (T_P, M_P)$ s.t. $P_i = P_{i-1} \cup \{t_i\}$, $P_{i-1} \triangleright_{t_i} P_i$ and* exploreConsistentPrefixes$(\hat{h}, P_i)$ *returns* $\mathtt{true}$*; where $|T_P| < i \leq |T|$ and $P_{|T_P|} = P$.*

Proof. Let $\hat{h}$ be a client history and $P = (T_P, M_P)$ be a consistent prefix s.t. exploreConsistentPrefixes$(\hat{h}, P)$ returns $\mathtt{true}$. We prove the result by induction on the number of transactions not present in T_P. The base case, when $|T_P| = |T|$, immediately holds as $T \setminus T_P = \emptyset$. Let us assume that the inductive hypothesis holds for any prefix containing k transactions and let us show that it also holds for every consistent prefix with $k-1$ transactions. Let us thus assume that $|T_P| = k - 1$. As exploreConsistentPrefixes$(\hat{h}, P)$ returns $\mathtt{true}$, it must reach line 5 in Algorithm 8. Hence, there must exist a transaction $t_k \in T \setminus T_P$ s.t. $P \triangleright_{t_k} (P \cup \{t_k\})$ and exploreConsistentPrefixes$(\hat{h}, P \cup \{t_k\})$ returns $\mathtt{true}$. By induction hypothesis on $P \cup \{t_k\} = (T_k, M_k)$, there exist a distinct transactions $t_i \in T \setminus T_k$ and a collection consistent prefixes P_i s.t. $P_i = P_{i-1} \cup \{t_i\}$, $P_{i-1} \triangleright_{t_i} P_i$ and exploreConsistentPrefixes$(\hat{h}, P_i)$ returns $\mathtt{true}$; where $k < i \leq |T|$ and $P_k = P \cup \{t_k\}$. Thus, the inductive step holds thanks to prefix P_k. □

Lemma 4.5.6. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history and let pco be the relation defined as at line 2 in Algorithm 8. If* checkConsistency(h) *returns* $\mathtt{true}$*, there exists an extension $\hat{h} = (T, \mathsf{so}, \widehat{\mathsf{wr}})$ of h s.t. for every* $\mathtt{read}$ *event r, variable x and transaction t, (1) if $(t, r) \in \widehat{\mathsf{wr}}_x \setminus \mathsf{wr}_x$ then $t \in 0_x^r(\mathsf{pco})$, (2) if $\widehat{\mathsf{wr}}_x^{-1}(r) \uparrow$ then $1_x^r(\mathsf{pco}) = \emptyset$, and (3)* exploreConsistentPrefixes$(\hat{h}, \emptyset)$ *returns* $\mathtt{true}$.

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history s.t. checkConsistency(h) returns $\mathtt{true}$ and let pco, E_h, X_h be the objects described in lines 2-4 in Algorithm 7. If there exists a pair $(r, x) \in E_h$ for which $0_x^r(\mathsf{pco}) = \emptyset$, checkConsistency$(h)$ returns $\mathtt{false}$. Hence, E_h is empty if and only if X_h is empty. If $E_h = \emptyset$, Algorithm 7 executes line 7. Thus, taking $\hat{h} = h$, conditions (1), (2) and (3) trivially hold. Otherwise, Algorithm 7 executes line 8. Once again, as exploreConsistentPrefixes$(h, \emptyset)$ returns $\mathtt{true}$, there must exists $f \in X_h$ s.t. exploreConsistentPrefixes$(\hat{h}, \emptyset)$ returns $\mathtt{true}$; where $\hat{h} = \bigoplus_{(r,x) \in E_h} \mathsf{wr}_x(f(r, x), r)$. Thanks to the definition of f and $\hat{h}$ conditions (1), (2) and (3) are satisfied. □

Lemma 4.5.7. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history whose isolation configuration is composed of* $\{\mathtt{SER}, \mathtt{SI}, \mathtt{PC}, \mathtt{RC}\}$ *isolation levels. If Algorithm 8 returns* $\mathtt{true}$*, h is consistent.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history s.t. CHECKCONSISTENCY(h) returns $\mathtt{true}$ and let pco, E_h and X_h be defined as at lines 2-4 in Algorithm 7. By Lemma 4.5.6, there exists an extension of h $\hat{h} = (T, \mathsf{so}, \hat{\mathsf{wr}})$ s.t. for every $\mathtt{read}$ event r, variable x and transaction t, (1) if $\hat{\mathsf{wr}}_x^{-1}(r) \uparrow$ then $\mathtt{1}_x^r(\mathsf{pco}) = \emptyset$, (2) if $(t, r) \in \hat{\mathsf{wr}}_x \setminus \mathsf{wr}_x$ then $t \in \mathtt{0}_x^r(\mathsf{pco})$ and (3) EXPLORECONSISTENTPREFIXES$(\hat{h}, \emptyset)$ returns $\mathtt{true}$. By Lemma 4.5.5 applied on $\hat{h}$ and $\emptyset$, there exist distinct transactions $t_i \in T$ and a collection of prefixes of h, $P_i = (T_i, M_i)$, s.t. $P_i = P_{i-1} \cup \{t_i\}$, $P_{i-1} \triangleright_{t_i} P_i$ and EXPLORECONSISTENTPREFIXES$(\hat{h}, P_i)$ returns $\mathtt{true}$; where $P_0 = \emptyset$ and $0 < i \leq |T|$. Let co be the total order based on the aforementioned transactions t_i, i.e. $\mathsf{co} = \{(t_i, t_j) \mid i < j\}$. We construct a full history that extends $\hat{h}$ employing co and taking into account the isolation level of each transaction.

For every $\mathtt{read}$ event r, key x and visibility relation $\mathsf{v} \in \mathsf{vis}(\mathsf{iso}(h)(\mathsf{tr}(r)))$, let t_v, t_x^r be the transactions defined as follows:

$$t_\mathsf{v}^x = \max_{\mathsf{co}}\{t' \in T \mid t' \text{ writes } x \ \land\ \mathsf{v}(\mathsf{co})(t', r, x)\}$$
$$t_x^r = \max_{\mathsf{co}}\{t_\mathsf{v}^x \mid \mathsf{v} \in \mathsf{vis}(\mathsf{iso}(h)(\mathsf{tr}(r)))\} \tag{4.7}$$

Note that if v is a visibility relation associated to an axiom from $\mathtt{SER}, \mathtt{SI}, \mathtt{PC}, \mathtt{RA}$ and $\mathtt{RC}$ isolation levels, transactions t_v^x and t_x^r are well-defined as $\mathsf{v}(\mathtt{init}, r, x)$ holds. Thus, let $\overline{\mathsf{wr}}_x = \hat{\mathsf{wr}}_x \cup \{(t_x^r, r) \mid \hat{\mathsf{wr}}_x^{-1}(r) \uparrow\}$ and $\overline{\mathsf{wr}} = \bigcup_{x \in \mathsf{Keys}} \overline{\mathsf{wr}}_x$. As $\overline{\mathsf{wr}}_x^{-1}$ is a total function and $\overline{\mathsf{wr}}_x^{-1}(r)$ writes x we can conclude that $\overline{h} = (T, \mathsf{so}, \overline{\mathsf{wr}})$ is a full history.

We prove that $\overline{h}$ is also a witness of h. For that, we show that for every read event r, every key x and every transaction t, if $(t, r) \in \overline{\mathsf{wr}}_x \setminus \mathsf{wr}_x$, $t \in \mathtt{0}_x^r(\mathsf{pco})$. Two cases arise: $(t, r) \in \hat{\mathsf{wr}}_x \setminus \mathsf{wr}_x$ and $(t, r) \in \overline{\mathsf{wr}}_x \setminus \hat{\mathsf{wr}}_x$. The first case is quite straightforward, as if $(t, r) \in \hat{\mathsf{wr}}_x \setminus \mathsf{wr}_x$, by Property (1) of Lemma 4.5.6, $t \in \mathtt{0}_x^r(\mathsf{pco})$. The second case, $(t, r) \in \overline{\mathsf{wr}}_x \setminus \hat{\mathsf{wr}}_x$, is slightly more subtle. First, for every isolation level considered, if $(t, r) \in \overline{\mathsf{wr}}_x$ then $(t, \mathsf{tr}(r)) \in \mathsf{pco}$. Next, as CHECKCONSISTENCY(h) returns $\mathtt{true}$, the condition at line 5 does not hold. Hence, as pco is acyclic, we deduce that $(\mathsf{tr}(r), t) \notin \mathsf{pco}$. In addition, as $(t, r) \in \overline{\mathsf{wr}}_x \setminus \hat{\mathsf{wr}}_x$, $\hat{\mathsf{wr}}_x^{-1}(r) \uparrow$. By Property (2) of Lemma 4.5.6 employed during $\hat{h}$'s construction, we deduce that $\mathtt{1}_x^r(\mathsf{pco}) = \emptyset$. In conclusion, as $(\mathsf{tr}(r), t) \notin \mathsf{pco}$ and $\mathtt{1}_x^r(\mathsf{pco}) = \emptyset$, we conclude that $t \in \mathtt{0}_x^r(\mathsf{pco})$.

Finally, we prove that co witnesses that $\overline{h}$ is consistent. Let r be a $\mathtt{read}$ event, x be a key and t_1, t_2 be transactions s.t. $(t_1, r) \in \overline{\mathsf{wr}}_x$ and t_2 writes x. We prove that if there exists $\mathsf{v} \in \mathsf{vis}(\mathsf{iso}(\overline{h})(\mathsf{tr}(r)))$ s.t. $\mathsf{v}(\mathsf{co})(t_2, r, x)$ holds in $\overline{h}$ then $(t_2, t_1) \in \mathsf{co}$; which by Definition 4.3.1, we know it implies that $\overline{h}$ is consistent. Note that if $(t_1, r) \in \overline{\mathsf{wr}} \setminus \hat{\mathsf{wr}}$, by definition of t_x^r the statement immediately holds; so we can assume without loss of generality that $(t_1, r) \in \hat{\mathsf{wr}}_x$.

First, we note that proving that whenever $\mathsf{v}(\mathsf{co})(t_2, r, x)$ holds in $\overline{h}$, then $(t_2, t_1) \in \mathsf{co}$ is equivalent to prove that whenever $\mathsf{v}(\mathsf{co})(t_2, r, x)$ holds in $\overline{h}$, then $t_1 \notin T_{i-1}$; where i is the index of the transaction in T s.t. $t_2 = t_i$.

For every $i, 1 \leq i \leq |T|$ $P_{i-1} \triangleright_{t_i} P_i$, $\mathsf{so} \cup \hat{\mathsf{wr}} \subseteq \mathsf{co}$. Thus, by Definition 4.3.1, it suffices to show that for every read event r, $C_{\mathsf{iso}(h)(\mathsf{tr}(r))}(\mathsf{pco})(r)$ holds. For that, let $\hat{\mathsf{pco}} = \mathsf{FIX}(\lambda R : \text{SATURATE}(\hat{h}, R))(\mathsf{so} \cup \hat{\mathsf{wr}})^+$ be the partial commit order implied by $\hat{h}$.

As $\mathsf{iso}(h)$ is composed of $\{\mathtt{SER}, \mathtt{SI}, \mathtt{PC}, \mathtt{RA}, \mathtt{RC}\}$ isolation levels and $P_{i-1} \triangleright_{t_2} P_i$, by Property 2 of Definition 4.5.1, it suffices to prove that whenever $\mathsf{v}(\mathsf{co})(t_2, r, x)$, if $v \neq \mathsf{Conflict}$ then $v(\hat{\mathsf{pco}}_{t_2}^{P_i})(t, r, x)$ holds in $\hat{h}$, while if $v = \mathsf{Conflict}$, that there exists $t' \in T_{i-1}$ s.t. $v(\hat{\mathsf{pco}}_{t_2}^{P_i})(t', r, x)$

holds in $\hat{h}$; where $\hat{\mathsf{pco}}_{t_2}^{P_i}$ is obtained by applying Table 4.1 on $\hat{\mathsf{pco}}$. We analyze five different cases:

- $\underline{\mathsf{iso}(\overline{h})(\mathsf{tr}(r)) = \mathtt{SER}}$: In this case, $\mathsf{Serializability}(\mathsf{co})(t_2, r, x)$ holds in $\overline{h}$ if and only if $(t_2, \mathsf{tr}(r)) \in \mathsf{co}$. As $\hat{\mathsf{pco}}_{t_2}^{P_i}$ totally orders t_2 and every other transaction in T and $\hat{\mathsf{pco}}_{t_2}^{P_i} \subseteq \mathsf{co}$, we deduce that $(t_2, \mathsf{tr}(r)) \in \hat{\mathsf{pco}}_{t_2}^{P_i}$. Hence, $\mathsf{Serializability}(\hat{\mathsf{pco}}_{t_2}^{P_i})(t_2, r, x)$ holds in $\hat{h}$.

- $\underline{\mathsf{iso}(\overline{h})(\mathsf{tr}(r)) = \mathtt{SI}}$: Two disjoint sub-cases arise:

  - $\underline{\mathsf{Conflict}(\mathsf{co})(t_2, r, x) \text{ holds in } \overline{h}}$: This happens if and only if there exists a transaction t_3 and a key $y \in \mathsf{Keys}$ s.t. t_3 writes y, $\mathsf{tr}(r)$ writes y, $(t_2, t_3) \in \mathsf{co}^*$ and $(t_3, \mathsf{tr}(r)) \in \mathsf{co}$. Let j be the index s.t. $t_3 = t_j$. Then, as $\hat{\mathsf{pco}}_{t_3}^{P_j}$ totally orders t_3 and every other transaction and $\hat{\mathsf{pco}}_{t_3}^{P_j} \subseteq \mathsf{co}$, $(t_2, t_3) \in (\hat{\mathsf{pco}}_{t_3}^{P_j})^*$ and $(t_3, \mathsf{tr}(r)) \in \hat{\mathsf{pco}}_{t_3}^{P_j}$. Thus, $\mathsf{Conflict}(\hat{\mathsf{pco}}_{t_3}^{P_j})(t_2, r, x)$ holds in $\hat{h}$.
  - $\underline{\mathsf{Prefix}(\mathsf{co})(t_2, r, x) \text{ holds in } \overline{h} \text{ but } \mathsf{Conflict}(\mathsf{co})(t_2, r, x) \text{ does not}}$: We observe that $\mathsf{Prefix}(\mathsf{co})(t_2, r, x)$ holds in $\overline{h}$ if there exists a transaction t_3 s.t. $(t_2, t_3) \in \mathsf{co}^*$ and $(t_3, \mathsf{tr}(r)) \in \mathsf{so} \cup \overline{\mathsf{wr}}$. If $(t_3, \mathsf{tr}(r)) \in \overline{\mathsf{wr}} \setminus (\mathsf{so} \cup \hat{\mathsf{wr}})$, by Equation (4.7) there exist $y \in \mathsf{Keys}$ and $v \in \mathsf{vis}(\mathtt{SI})$ s.t. $v(\mathsf{co})(t_3, r, y)$ holds in $\hat{h}$. Note that $v \neq \mathsf{Conflict}$ as otherwise $\mathsf{Conflict}(\mathsf{co})(t_2, r, x)$ would hold in $\overline{h}$. Hence, $v = \mathsf{Prefix}$ and by transitivity of co, we conclude that $\mathsf{Prefix}(\mathsf{co})(t_2, r, x)$ holds in $\hat{h}$. As $\hat{\mathsf{pco}}_{t_2}^{P_i}$ totally orders t_2 with respect every other transaction in t_2 and $\hat{\mathsf{pco}}_{t_2}^{P_i} \subseteq \mathsf{co}$, we conclude that $\mathsf{Prefix}(\hat{\mathsf{pco}}_{t_2}^{P_i})(t_2, r, x)$ holds in $\hat{h}$.

- $\underline{\mathsf{iso}(\overline{h})(\mathsf{tr}(r)) = \mathtt{PC}}$: In this case, $\mathsf{Prefix}(\mathsf{co})(t_2, r, x)$ holds in $\overline{h}$ if and only if there exists a transaction t_3 s.t. $(t_2, t_3) \in \mathsf{co}^*$ and $(t_3, \mathsf{tr}(r)) \in \mathsf{so} \cup \overline{\mathsf{wr}}$. If $(t_3, \mathsf{tr}(r)) \in \overline{\mathsf{wr}} \setminus (\mathsf{so} \cup \hat{\mathsf{wr}})$, by Equation (4.7) there exist $y \in \mathsf{Keys}$ and $v \in \mathsf{vis}(\mathtt{SI})$ s.t. $v(\mathsf{co})(t_3, r, y)$ holds in $\hat{h}$. Hence, by transitivity of co, we conclude that $\mathsf{Prefix}(\mathsf{co})(t_2, r, x)$ holds in $\hat{h}$. As $\hat{\mathsf{pco}}_{t_2}^{P_i}$ totally orders t_2 with respect every other transaction in t_2 and $\hat{\mathsf{pco}}_{t_2}^{P_i} \subseteq \mathsf{co}$, we conclude that $\mathsf{Prefix}(\hat{\mathsf{pco}}_{t_2}^{P_i})(t_2, r, x)$ holds in $\hat{h}$.

- $\underline{\mathsf{iso}(\overline{h})(\mathsf{tr}(r)) = \mathtt{RA}}$: In this case, $\mathsf{Read\ Atomic}(\mathsf{co})(t_2, r, x)$ holds in $\overline{h}$ if and only if $(t_2, \mathsf{tr}(r)) \in \mathsf{so} \cup \overline{\mathsf{wr}}$. We observe that by Equation (4.7), if $(t_2, \mathsf{tr}(r)) \in \overline{\mathsf{wr}} \setminus (\mathsf{so} \cup \hat{\mathsf{wr}})$, then $t_2 = t_x^r$ and $\mathsf{Read\ Atomic}(\hat{\mathsf{pco}}_{t_2}^{P_i})(t_2, r, x)$ holds in $\hat{h}$. Hence, $(t_2, \mathsf{tr}(r)) \in \mathsf{so} \cup \hat{\mathsf{wr}}$; which is a contradiction. Thus, as $(t_2, \mathsf{tr}(r)) \in \mathsf{so} \cup \hat{\mathsf{wr}}$, $\mathsf{Read\ Atomic}(\hat{\mathsf{pco}}_{t_2}^{P_i})(t_2, r, x)$ holds in $\hat{h}$.

- $\underline{\mathsf{iso}(\overline{h})(\mathsf{tr}(r)) = \mathtt{RC}}$: Similarly to the previous case, we observe that the formula $\mathsf{Read\ Committed}(\mathsf{co})(t_2, r, x)$ holds in $\overline{h}$ iff $(t_2, \mathsf{tr}(r)) \in (\mathsf{so} \cup \overline{\mathsf{wr}}); \mathsf{po}^*$. We observe that by Equation (4.7), if $(t_2, \mathsf{tr}(r)) \in \overline{\mathsf{wr}} \setminus (\mathsf{so} \cup \hat{\mathsf{wr}}); \mathsf{po}^*$, then $t_2 = t_x^r$ and $\mathsf{Read\ Committed}(\hat{\mathsf{pco}}_{t_2}^{P_i})(t_2, r, x)$ holds in $\hat{h}$. Therefore, $(t_2, r) \in \mathsf{so} \cup \hat{\mathsf{wr}}; \mathsf{po}^*$; which is a contradiction. Thus, as $(t_2, \mathsf{tr}(r)) \in \mathsf{so} \cup \hat{\mathsf{wr}}; \mathsf{po}^*$, $\mathsf{Read\ Committed}(\hat{\mathsf{pco}}_{t_2}^{P_i})(t_2, r, x)$ holds in $\hat{h}$.

□

4.5.2 Sufficient Conditions for Checking Consistency of Client Histories in Polynomial Time

In general, Algorithm 7 is exponential the number of conflicts in h. The number of *conflicts* is denoted by $\#\mathtt{conf}(h)$. The number of conflicts exponent is implied by the number of mappings in X_h explored by Algorithm 7 (E_h is the set of conflicts in h). The history width and size exponents comes from the number of prefixes explored by Algorithm 8 which is $|h|^{\mathtt{width}(h)} \cdot \mathtt{width}(h)^{|\mathsf{Keys}|}$ in the worst case (prefixes can be equivalently described by a set of so-maximal transactions and a mapping associating keys to sessions).

Theorem 4.5.8. *For every client history h whose isolation configuration is composed of $\{\mathtt{SER}, \mathtt{SI}, \mathtt{PC}, \mathtt{RA}, \mathtt{RC}\}$ isolation levels, Algorithm 7 runs in $\mathcal{O}(|h|^{\#\mathtt{conf}(h)+\mathtt{width}(h)+9} \cdot \mathtt{width}(h)^{|\mathsf{Keys}|})$. Moreover, if no transaction employs $\mathtt{SI}$ isolation level, Algorithm 7 runs in $\mathcal{O}(|h|^{\#\mathtt{conf}(h)+\mathtt{width}(h)+8})$.*

On bounded, conflict-free histories only using $\mathtt{SER}, \mathtt{PC}, \mathtt{RA}, \mathtt{RC}$ as isolation levels, Algorithm 7 runs in polynomial time. For instance, standard reads and writes can be simulated using `INSERT` and `SELECT` with `WHERE` clauses that select rows based on their key being equal to some particular value. In this case, histories are conflict-less (wr would be defined for the particular key asked by the clause, and writes on other keys would not satisfy the clause). A more general setting where `WHERE` clauses restrict only values that are immutable during the execution (e.g., primary keys) and deletes only affect non-read rows also falls in this category.

The proof of Theorem 4.5.8 is split in two Lemmas: Lemma 4.5.12 analyzes the complexity of Algorithm 8 while Lemma 4.5.13 relies on the previous result to conclude the complexity of Algorithm 7.

Algorithm 9 Checking if $P \triangleright_t (P \cup \{t\})$ holds in h

```
1: function isConsistentExtension($h = (T, \mathsf{so}, \mathsf{wr})$, $P = (T_P, M_P)$, $t$)
                                                        ▷ We assume $t \notin T_P$.
2:     let $\mathsf{pco} = \mathsf{FIX}(\lambda R : \text{SATURATE}(h, R))(\mathsf{so} \cup \mathsf{wr})^+$
3:     if $\exists t' \in T \setminus T_P$ s.t. $(t', t) \in \mathsf{pco}$ then        ▷ Condition 1
4:         return false
5:     for all $r \in \mathsf{reads}(h)$ s.t. $\mathsf{tr}(r) \notin T_P \cup \{t\}, v \in \mathsf{vis}(\mathsf{iso}(h))(\mathsf{tr}(r))$ do
6:         if $\mathsf{vp}_v^P(t, r)$ does not hold in $h$ then        ▷ Condition 2
7:             return false
8:     return true
```

Lemma 4.5.9. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history, $P = (T_P, M_P)$ be a consistent prefix of h and $t \in T \setminus T_P$ be a transaction. Algorithm 9 returns* true *if and only if $P \triangleright_t (P \cup \{t\})$.*

Proof. Clearly, $P \cup \{t\}$ is an extension of P.isConsistentExtension(h, P, t) returns true if and only if conditions at lines 3 and lines 5 in Algorithm 9 hold. This is equivalent to respectively satisfy Properties 1 and 2 of Definition 4.5.1. By Definition 4.5.1, this is equivalent to $P \triangleright_t (P \cup \{t\})$. □

Lemma 4.5.10. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history and $k \in \mathbb{N}$ be a bound in $\mathsf{iso}(h)$. For any consistent prefix $P = (T_P, M_P)$ of h and any transaction $t \in T \setminus T_P$, Algorithm 9 runs in $\mathcal{O}(|h|^{k+3})$.*

Proof. We analyze the cost of Algorithm 9. First, as $\mathsf{pco} \subseteq T \times T$, by Lemma 4.4.4, line 2 runs in $\mathcal{O}(|h|^2 \cdot |h|^{k+1})$. Next, the condition at line 3 can be checked in $\mathcal{O}(|T|)$. Finally, the condition at line 5 can be checked in $\mathcal{O}(|T| \cdot k \cdot U)$; where U is an upper-bound on the complexity of checking $\mathsf{vp}^P_v(t, r)$. With the aid of Lemma 4.4.3, we deduce that $U \in \mathcal{O}(|h|^{k-2})$. Altogether, we conclude that Algorithm 9 runs in $\mathcal{O}(|h|^{k+3})$. □

Lemma 4.5.11. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history. If $\mathsf{iso}(h)$ is composed of $\{\mathtt{SER}, \mathtt{SI}, \mathtt{PC}, \mathtt{RA}, \mathtt{RC}\}$ isolation levels, then 5 is a bound of $\mathsf{iso}(h)$. Moreover, if no transaction has $\mathtt{SI}$ as isolation, 4 is a bound on $\mathsf{iso}(h)$.*

Proof. Let h be a history as described in the hypothesis. First, all isolation levels in the set $\{\mathtt{SER}, \mathtt{SI}, \mathtt{PC}, \mathtt{RA}, \mathtt{RC}\}$ employ at most two axioms. Moreover, every axiom described employs at most 5 quantifiers: three universal quantifiers and at most two existential quantifiers. Hence, 5 is a bound on $\mathsf{iso}(h)$. Note that $\mathsf{Conflict}$ is the only axiom employing two existential quantifiers; so if no transaction employs $\mathtt{SI}$, 4 bounds $\mathsf{iso}(h)$. □

Lemma 4.5.12. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history whose isolation configuration is composed of $\{\mathtt{SER}, \mathtt{SI}, \mathtt{PC}, \mathtt{RA}, \mathtt{RC}\}$ isolation levels. Algorithm 8 runs in $\mathcal{O}(|h|^{\mathtt{width}(h)+9} \cdot \mathtt{width}(h)^{|\mathsf{Keys}|})$. Moreover, if no transaction has $\mathtt{SI}$ as isolation level, Algorithm 8 runs in $\mathcal{O}(|h|^{\mathtt{width}(h)+8})$.*

Proof. For proving the result, we focus only on prefixes that are computable by Algorithm 8. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history. A prefix P of h is *computable* if either $P = \emptyset$ or there exist a transaction t and a prefix P' s.t. $P = P' \cup \{t\}$ and P' is computable.

Intuitively, computable prefixes represent recursive calls of Algorithm 8 when employed by Algorithm 7. Indeed, Algorithm 7 only employs Algorithm 8 at lines 7 and 11. In both cases, $P' = \emptyset$ is the initial call to Algorithm 8. Moreover, the condition at line 3 justifies the recursive definition.

On one hand, we observe that any call to Algorithm 8 is associated to a computable prefix and on the other hand, Algorithm 8 does not explore two equivalent computable prefix thanks to the global variable **seen** (line 4). Therefore, Algorithm 8 runs in $\mathcal{O}(N \cdot U)$; where N is the number of distinct equivalence class of prefixes of h and U is an upper-bound on the running time of Algorithm 8 on a fixed prefix without doing any recursive call.

We first compute an upper-bound of N. For any computable prefix P, we can deduce by induction on the length of P that there exists transactions $t_i \in T_P$ and a collection of computable prefixes of h, $P_i = (T_i, M_i)$ and transactions t_i s.t. $P_{|T_P|} = P$, $P_i = P_{i-1} \cup \{t_i\}$ and $P_{i-1} \triangleright_{t_i} P_i$; where $P_0 = \emptyset$ and $0 < i \leq |T_P|$. The base case is immediate as $|T_P| = 0$ implies that $T' = \emptyset$ while the inductive step can be simply obtained by applying the recursive definition of computable prefix.

Let $P = (T_P, M_P)$ be in what follows a computable prefix of h. We observe that both T_P and M_P are determined by its so-maximal transactions. Let $t, t' \in T$ be a pair of transactions s.t. $(t, t') \in \mathsf{so}$ and $t' \in T_P$. As $t' \in T_P$ there must exist an index $i, 1 \leq i \leq |T_P|$ s.t. $P_i = P_{i-1} \cup \{t'\}$. Therefore, as $P_{i-1} \triangleright_{t'} P_{i-1}$, $t \in T_{i-1} \subseteq T_P$. In particular, if t' is a so-maximal transaction in T_P, all its so-predecessors are also contained in T_P; and hence, T_P can be characterized by its so-maximal transactions. Moreover, by induction on the length of P we can prove that for every key x, $M_P(x)$ is a so-maximal transaction: the base case, $|T_P| = 0$ is immediate while the inductive step is obtained by the definition of $P \cup \{t''\}, t'' \notin T_P$. Hence, the number of computable prefixes of a history is in $\mathcal{O}(|T|^{\texttt{width}(h)} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$. Thus, $N \in \mathcal{O}(|h|^{\texttt{width}(h)} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$. Moreover, if no transaction employs $\texttt{SI}$ as isolation level, prefixes with identical transaction set coincide. Hence, in such case, $N \in \mathcal{O}(|h|^{\texttt{width}(h)})$.

We conclude the proof bounding U. If $|T_P| = |T|$, Algorithm 8 runs in $\mathcal{O}(1)$; so we can assume without loss of generality that $|T_P| \neq |T|$. In such case, U represent the cost of executing lines 3-7 in Algorithm 8. Thus, $U \in \mathcal{O}((|T| - |T_P|) \cdot V)$; where V is the cost of checking $P \triangleright_t (P \cup \{t\})$ for a transaction $t \in T \setminus T_P$. By Lemma 4.5.9, Algorithm 9 can check if $P \triangleright_t (P \cup \{t\})$ and thanks to Lemma 4.5.10, Algorithm 9 runs in $\mathcal{O}(|h|^{k+3})$; where k is a bound on $\mathsf{iso}(h)$. Thus, $U \in \mathcal{O}(|h|^{k+4})$.

Thanks to Lemma 4.5.11, we conclude that Algorithm 8 runs in $\mathcal{O}(|h|^{\texttt{width}(h)+9} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$ and, if no transaction employs $\texttt{SI}$ as isolation level, then it runs in $\mathcal{O}(|h|^{\texttt{width}(h)+8})$. □

Lemma 4.5.13. *Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a client history whose isolation configuration is composed of $\{\texttt{SER}, \texttt{SI}, \texttt{PC}, \texttt{RA}, \texttt{RC}\}$ isolation levels. Algorithm 7 runs in $\mathcal{O}(|h|^{\#\texttt{conf}(h)+\texttt{width}(h)+9} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$. Moreover, if no transaction has $\texttt{SI}$ as isolation level, Algorithm 7 runs in $\mathcal{O}(|h|^{\#\texttt{conf}(h)+\texttt{width}(h)+8})$.*

Proof. Let $h = (T, \mathsf{so}, \mathsf{wr})$ be a history satisfying the hypothesis of the Lemma. We decompose our analysis in two sections, the first one where we analyze the complexity of executing lines 2-4 and second one where we analyze the complexity of executing lines 5-12. We observe that by Lemma 4.5.11, 5 is a bound on $\mathsf{iso}(h)$.

In line 2, Algorithm 7 computes pco. On one hand, computing $(\mathsf{so} \cup \mathsf{wr})^+$ is in $\mathcal{O}(|T|^3)$. On the other hand, as $\mathsf{pco} \subseteq T \times T$ and by Lemma 4.4.4, executing $\textsc{saturate}(h, (\mathsf{so} \cup \mathsf{wr})^+)$ is in $\mathcal{O}(|h|^6)$; we deduce that computing pco after compute $(\mathsf{so} \cup \mathsf{wr})^+$ is in $\mathcal{O}(|h|^8)$.

In line 3, Algorithm 7 computes E_h. As wr is acyclic, for a given key x and transaction t, $\texttt{value}_{\mathsf{wr}}(t, x) \in \mathcal{O}(|T|)$. Therefore, computing $\mathtt{1}_x^r(\mathsf{pco})$ is in $\mathcal{O}(|T|)$ as we assume that for every $r \in \mathsf{Vals}$, computing $\texttt{WHERE}(r)(v) \in \mathcal{O}(1)$. Thus, computing E_h is in $\mathcal{O}(|h|^3)$.

Finally, in line 4, Algorithm 7 computes X_h. Note that X_h can be seen is a $\bigtimes_{(r,x) \in E_h} \mathtt{0}_x^r(\mathsf{pco})$. Computing each $\mathtt{0}_x^r(\mathsf{pco})$ set is in $\mathcal{O}(|T|)$; so computing all of them is in $\mathcal{O}(|T| \cdot |E_h|)$. As each set $\mathtt{0}_x^r(\mathsf{pco})$ is a subset of T, applying the cartesian-product definition of X_h we can compute X_h in $\mathcal{O}(|T|^{|E_h|})$. Therefore, as $|E_h| = \#\texttt{conf}(h)$, we conclude that computing X_h is in $\mathcal{O}(|h| \cdot \#\texttt{conf}(h) + |h|^{\#\texttt{conf}(h)})$ and that $|X_h| \in \mathcal{O}(|h|^{\#\texttt{conf}(h)})$. Altogether, as $\#\texttt{conf}(h) \leq |T|^2$, we deduce that computing lines 2-4 of Algorithm 7 is in $\mathcal{O}(|h|^8 + |h|^{\#\texttt{conf}(h)})$.

Next, we analyze the complexity of executing lines 5-12. Four disjoint cases arise, one per boolean condition in Algorithm 7. The first one, checking if pco is cyclic (line 5), is in $\mathcal{O}(|h|)$. The second one, checking if $\exists (r, x) \in E_h$ s.t. $\mathtt{0}_x^r(\mathsf{pco}) = \emptyset$ (line 6), clearly runs

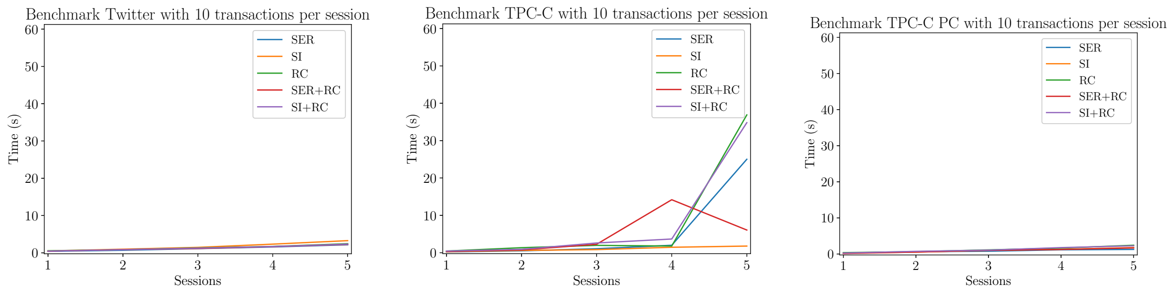


Figure 4.13: Running time of Algorithm 7 while increasing the number of sessions. Each point represents the average running time of 5 random clients of such size.

in $\mathcal{O}(\#\texttt{conf}(h) \cdot |h|)$. The third one, checking if $E_h = \emptyset$ and executing Algorithm 8 is in $\mathcal{O}(\#\texttt{conf}(h) + |h|^{\texttt{width}(h)+9} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$ thanks to Lemma 4.5.12.

Finally we analyze the last case, computing an extension of h for each mapping in X_h and then executing Algorithm 7 (lines 8-12). On one hand, computing each history is in $\mathcal{O}(|h|^3)$ as we require to define both $\mathsf{so} \subseteq T \times T$ and $\mathsf{wr} \subseteq \mathsf{Keys} \times T \times T$. On the other hand, as the size of each extension of h is in $\mathcal{O}(|h|)$, executing Algorithm 7 for a given history is in $\mathcal{O}(|X_h| \cdot |h|^{\texttt{width}(h)+9} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$ thanks again to Lemma 4.5.12. Altogether, for each mapping $f \in X_h$, executing lines 10-11 is in $\mathcal{O}(|h|^{\texttt{width}(h)+9} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$. As $|X_h| \in \mathcal{O}(|h|^{\#\texttt{conf}(h)})$, we conclude that executing this last case is in $\mathcal{O}(|h|^{\#\texttt{conf}(h)+\texttt{width}(h)+9} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$.

We then conclude that Algorithm 7 runs in $\mathcal{O}(|h|^{\#\texttt{conf}(h)+\texttt{width}(h)+9} \cdot \texttt{width}(h)^{|\mathsf{Keys}|})$. Moreover, if no transaction employs `SI` as isolation level, Lemma 4.5.12 allows us to deduce that in such case Algorithm 7 runs in $\mathcal{O}(|h|^{\#\texttt{conf}(h)+\texttt{width}(h)+9})$. $\square$

4.6 Experimental Evaluation

We evaluate an implementation of CHECKCONSISTENCY in the context of the Benchbase [51] database benchmarking framework. We apply this algorithm on histories extracted from randomly generated client programs of a number of database-backed applications. We use PostgreSQL 14.10 as a database. The experiments were performed on an Apple M1 with 8 cores and 16 GB of RAM.

Implementation. We extend the Benchbase framework with an additional package for generating histories and checking consistency. Applications from Benchbase are instrumented in order to be able to extract histories, the wr relation in particular. Our implementation is publicly available [33].

Our tool takes as input a configuration file specifying the name of the application and the isolation level of each transaction in that application. For computing the wr relation and generating client histories, we extend the database tables with an extra column `WRITEID` which is updated by every `write` instruction with a unique value. SQL queries are also modified to return whole rows instead of selected columns. To extract the wr relation for `UPDATE` and `DELETE` we add `RETURNING` clauses. Complex operators such as `INNER JOIN` are substituted by simple juxtaposed SQL queries (similarly to [30]). We map the result of each query to local structures for generating the corresponding history. Transactions aborted by the database (and not explicitly by the application) are discarded.

Benchmark. We analyze a set of benchmarks inspired by real-world applications and evaluate them under different types of clients and isolation configurations. We focus on isolation configurations implemented in PostgreSQL, i.e. compositions of `SER`, `SI` and `RC` isolation levels.

In average, the ratio of SER/SI transactions is 11% for Twitter and 88% for TPC-C and TPC-C PC. These distributions are obtained via the random generation of client programs implemented in BenchBase. In general, we observe that the bottleneck is the number of possible history extensions enumerated at line 9 in Alg. 3 and not the isolation configuration. This number is influenced by the distribution of types of transactions, e.g., for TPC-C, a bigger number of transactions creating new orders increases the number of possible full history extensions. We will clarify.

Twitter [51] models a social network that allows users to publish tweets and get their followers, tweets and tweets published by other followers. We consider five isolation configurations: `SER`, `SI` and `RC` and the heterogeneous `SER` + `RC` and `SI` + `RC`, where publishing a tweet is `SER` (resp., `SI`) and the rest are `RC`. The ratio of `SER` (resp. `SI`) transactions w.r.t. `RC` is 11% on average.

TPC-C [101] models an online shopping application with five types of transactions: reading the stock, creating a new order, getting its status, paying it and delivering it. We consider five isolation configurations: the homogeneous `SER`, `SI` and `RC` and the combinations `SER` + `RC` and `SI` + `RC`, where creating a new order and paying it have `SER` (respectively `SI`) as isolation level while the rest have `RC`. The ratio of `SER` (resp. `SI`) transactions w.r.t. `RC` is 88% on average.

TPC-C PC is a variant of the TPC-C benchmark whose histories are always conflict-free. `DELETE` queries are replaced by `UPDATE` with the aid of extra columns simulating the absence of a row. Queries whose `WHERE` clauses query mutable values are replaced by multiple simple instructions querying only immutable values such as unique ids and primary keys.

Experimental Results. We designed two experiments to evaluate CHECKCONSISTENCY's performance for different isolation configurations increasing the number of transactions per session (the number of sessions is fixed), the number of sessions (the number of transactions per session is fixed), resp. We use a timeout of 60 seconds per history.

The first experiment investigates the scalability of Algorithm 7 when increasing the number of sessions. For each benchmark and isolation configuration, we consider 5 histories of random clients (each history is for a different client) with an increasing number of sessions and 10 transactions per session (around 400 histories across all benchmarks). No timeouts appear with less than 4 sessions. Figure 4.13 shows the running time of the experiment.

The second experiment investigates the scalability of Algorithm 7 when increasing the number of transactions. For each benchmark and isolation configuration, we consider 5 histories of random clients, each having 3 sessions and an increasing number of transactions per session (around 1900 histories across all benchmarks). Figure 4.14 shows its running time.

The runtime similarities between isolation configurations containing `SI` versus those without it show that in practice, the bottleneck of Algorithm 7 is the number of possible history extensions enumerated at line 11 in Algorithm 7; i.e. the number of conflicts in a history; not the isolation configuration considered. This number is influenced by the distribution of types of transactions, e.g., for TPC-C, a bigger number of transactions creating new orders in-

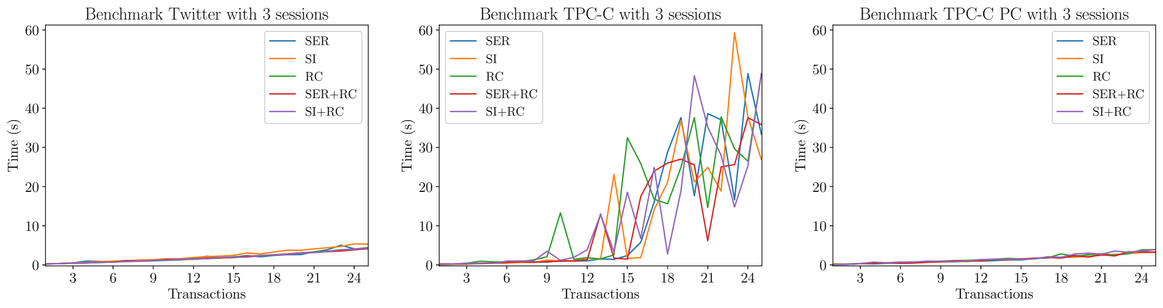


Figure 4.14: Running time of Algorithm 7 increasing the number of transactions per session. We plot the average running time of 5 random clients of such size.

creases the number of possible full history extensions. Other isolation levels not implemented by PostgreSQL, e.g., prefix consistency PC, are expected to produce similar results.

Both experiments show that Algorithm 7 scales well for histories with a small number of writes (like Twitter) or conflicts (like TPC-C PC). In particular, Algorithm 7 is quite efficient for typical workloads needed to expose bugs in production databases which contain less than 10 transactions [29, 79, 67].

A third experiment compares Algorithm 7 with a baseline consisting in a naive approach where we enumerate witnesses and executions of such witnesses until consistency is determined. We consider Twitter and TPC-C as benchmarks and execute 5 histories of random clients, each having 3 sessions and an increasing number of transactions per session (around 100 histories across all benchmarks). We execute each client under RC and check the obtained histories for consistency with respect to SER.

The naive approach either times out for 35.5%, resp., 95.5% of the histories of Twitter, resp., TPC-C, or finishes in 5s on average (max 25s). In comparison, Algorithm 7 has no timeouts for Twitter and times out for 5.5% of the TPC-C histories; finishing in 1.5s on average (max 12s). Averages are computed w.r.t. non-timeout instances. The total number of executed clients is around 100. Only one TPC-C history was detected as inconsistent, which shows that the naive approach does not timeout only in the worst-case (inconsistency is a worst-case because all extensions and commit orders must be proved to be invalid).

A similar analysis on the TPC-C PC benchmark is omitted: TPC-C PC is a conflict-free variation of TPC-C with more operations per transaction. Thus, the rate of timeouts in the naive approach increases w.r.t. TPC-C, while the rate of timeouts using Algorithm 7 decreases.

Comparisons with prior work [29, 13, 67, 79] are not possible as they do not apply to SQL (see Section 4.7 for more details).

This evaluation demonstrates that our algorithm scales well to practical testing workloads and that it outperforms brute-force search.

4.7 Related Work

The formalization of database isolation levels has been considered in previous work. Adya [9] has proposed axiomatic specifications for isolation levels, which however do not concern more modern isolation levels like PC or SI and which are based on low-level modeling of database snapshots. We follow the more modern approach in [45, 29] which however addresses the restricted case when transactions are formed of reads and writes on a *static* set of keys (variables) and not generic SQL queries, and all the transactions in a given execution have the same isolation level. Our axiomatic model builds on axioms defined by Biswas et al. [29] which are however applied on a new model of executions that is specific to SQL queries.

The complexity of checking consistency w.r.t isolation levels has been studied in [90, 29]. The work of Papadimitriou [90] shows that checking serializability is NP-complete while the work of Biswas et al. [29] provides results for the same isolation levels as in our work, but in the restricted case mentioned above.

Checking consistency in a non-transactional case, shared-memory or distributed systems, has been investigated in a number of works, e.g., [31, 58, 52, 43, 63, 55, 7, 57, 11]. Transactions introduce additional challenges that make these results not applicable.

Existing tools for checking consistency in the transactional case of distributed databases, e.g., [29, 13, 67, 79] cannot handle SQL-like semantics, offering guarantees modulo their transformations to reads and writes on static sets of keys. Our results show that handling the SQL-like semantics is strictly more complex (NP-hard in most cases).

5 Arbitration-Free Consistency is Available (and Vice Versa)

5.1 Introduction

In this chapter, we focus on characterizing which consistency models support available implementations of which storage systems.

A storage system is composed of a collection of objects that can be read or modified using a set of operations (the API of the storage). Specifications are expressed in terms of an abstract model of storage executions, which is defined as a set of binary relations among events—each event corresponding to an invocation of an operation on an object. These relations capture typical control-flow dependencies–such as invocations occurring at the same replica–data-flow dependencies–where certain updates affect the result of a query–and a total order used as a "tie-breaker" to fix an order between concurrent invocations. The latter is called *arbitration order* and it has an important role in our main result.

In a distributed storage system, implementations typically rely on communication protocols to share the effects of invocations among all replicas. They also use specific algorithms to merge the effects received from other replicas into the local replica's state. As a result, each invocation can be viewed as executing within a specific *context*–that is, the set of prior operations, including those received from remote replicas.

A storage specification defines the expected behavior of the system. It consists of two parts:

- a *consistency model*, restricting the possible contexts in which each invocation may execute.
- an *operation specification*, describing the allowable effects of an invocation, given its context.

A consistency model consists of a set of *visibility* formulas saying when an invocation belongs to the context of another invocation. This "being in the context of" binary relation is defined via combinations of the binary relations mentioned above (by standard composition, union, and transitive closure). For instance, a visibility formula may state that all prior invocations at the same replica should be included in the context. An *operation specification* consists of a set of functions that characterize the read and write behavior of an invocation, in particular, the value written by writes. Note that this value is not always fixed since we allow operations that read and write at the same time, e.g., Compare-and-Swap which writes a given object only if the old value equals some other value given as input. We also allow SQL transactions whose effects are even more complex.

We show that our framework covers many possible storage specifications, including Last-Writer-Wins and Multi-Value Key-Value stores, Key-Value stores with Compare-and-Swap operations, Key-Value stores with counters, as well as transactional and non-transactional SQL stores, and many possible consistency models including Return-Value Consistency, Causal Consistency, Sequential Consistency, and transactional isolation levels like Snapshot Isolation and Serializability.

The arbitration-free consistency (*AFC*) theorem. Our main result states roughly, that a storage system has an available implementation if and only if the visibility formulas that define its consistency model *exclude any meaningful use of the total arbitration order.* Such a consistency model is called *arbitration-free.* As in previous works, we consider an implementation to be *available* if operations can be answered immediately on every replica (without waiting for messages from other replicas).

The proof of the AFC theorem is quite challenging, one reason being the very expressive and abstract specification framework that we consider. Proving that there exist available implementations for arbitration-free consistency models is the easier part since arbitration-freeness implies that the model is weaker than causal consistency, and the latter supports available implementations [21, 80, 82, 22]. The opposite direction is much more difficult and is described in two stages.

We first consider a basic case, in which operations read and/or write a single value from/to a single object. This yields a reasonably simple proof, while still covering consistency models such as Return-Value Consistency, Causal Consistency, Prefix Consistency and Sequential Consistency, and objects such as a key-value store, with ordinary put / get operations or extended with Fetch-and-Add and Compare-and-Swap operations.

Then, we consider a general class of objects where operations can read and/or write multiple objects at the same time, and reads may compute their return value from multiple updates. In this very generic context, we need to introduce some number of restrictions (assumptions) which are however satisfied by all practical cases that we are aware of (see Section 5.8). This is to exclude pathological cases that arise from starting with a very abstract formal model.

To summarize, we provide the first characterization of distributed storage formal specifications that support available implementations which takes into account both consistency constraints and the semantics of the implemented objects. At a high level, the key insight behind our result is that in an asynchronous system, where replicas coordinate only through the exchange of messages, they can establish at most a causal order between operations. The arbitration order, in contrast, is total: it compares operations that are concurrent and therefore incomparable under causality. Determining such a total order would require additional synchronization between replicas, coordination that cannot be achieved in an always-available manner.

5.2 Motivating Examples

```
PUT(x,1);      || PUT(y,2);
a = GET(y);    || b = GET(x);
```

(a) Sequential Consistency and PUT, GET operations.

```
PUT(x, 1);     || PUT(y, 1);
PUT(x, 2);     || PUT(y, 2);
...            || ...
PUT(x, K);     || PUT(y, K);
a = GET(x);    || b = GET(y);
```

(b) Bounded Staleness and PUT, GET operations.

```
FAA(x, 1);     || FAA(x, 2);
```

(c) Sequential Consistency and FAA operations.

```
FAA(x, 1);     || FAA(y, 2);
FAA(y, 3);     || FAA(x, 4);
```

(d) Prefix Consistency and FAA operations.

Figure 5.1: Different litmus programs with two concurrent sessions showing the absence of available implementations for selected pairs of consistency models and operation specifications.

We illustrate the broad applicability of the AFC theorem through various storage specifications, each reflecting different trade-offs between consistency and operation semantics. We argue about the diversity of reasoning required and motivate the need for a unified framework.

As a starting point, we consider a standard key-value store with PUT and GET operations; PUT(x, v) writes the value v to object (key) x, and GET(x) reads the latest[1] value of object x. As consistency model, we consider the standard *Sequential Consistency* (SC) whose formalization uses arbitration to postulate an order in which different operations interleave. By the AFC theorem, the latter implies that there exists no available implementation that ensures SC. Intuitively, the proof is based on a *litmus* program like in Figure 5.1a. This program contains two concurrent sessions, each executed at a different replica. Also, x and y are initially 0. An SC available implementation should allow an execution in which, intuitively, the two replicas operate without exchanging any messages, resulting in both final get operations returning 0. However, this outcome violates sequential consistency, as it cannot be produced by any interleaving of the operations—leading to a contradiction.

We remark that this argument proves a version of the CAP theorem that is stronger than the one proved in [59]. The latter proof relies on the *real-time ordering* requirement that is embedded in linearizability — a consistency model stronger than sequential consistency (cf. [70]).

Such a proof can be generalized to the case where PUT / GET operations are replaced for instance, by ADD / CONTAINS operations on a set, i.e., PUT(x, v) and GET(x) in Figure 5.1a are replaced by ADD(x) and CONTAINS(x) (and similarly for operations on y). As in the previous case, an SC available implementation should allow an execution without exchange of messages, resulting in both final CONTAINS operations returning false (the set does not contain the element), which is an SC violation.

On the other hand, if we consider a weaker consistency model, a straightforward variation of the program in Figure 5.1a can not be used to prove non-existence of available implementations. For instance, consider *Bounded Staleness* [2] a weakening of SC, which requires that

[1]We assume a standard semantics based on the Last-Writer-Wins conflict resolution policy.

each get operation observes all preceding put operations (on the same object), except possibly the most recent $K - 1$, for some fixed value of K. The put operations are still required to execute following some fixed arbitration order as in SC (see Section 5.8.1 for a precise definition). This weakening for $K = 2$ admits an execution of t he program in Figure 5.1a where both final get operations return 0 (the get operations may miss the only put in the program). Therefore, this program cannot be used to show non-existence of available implementations. Instead, one can use the program given in Figure 5.1b, which contains K put operations in each session. One can follow now the same strategy as above and show that an execution without exchange of messages makes both get operations return 0, and this violates bounded staleness.

If we weaken consistency even further and consider *Causal Consistency* (CC) [96], then the AFC theorem will imply existence of available implementations (which is known [21, 80, 82, 22]).

Now, if we change the set of operations and consider a storage system with only Fetch-and-Add operations ($\mathtt{FAA}(x, v)$ returns the old value of x and adds v, atomically), then a proof for non-existence of SC available implementations can be done using the program in Figure 5.1c with only one FAA in each session. An execution without exchange of messages will imply that both FAA return the initial value of x, and this is a violation of SC.

If we weaken consistency to *Prefix Consistency* (PC) [42], then the previous program is not suitable. An execution where both FAA in Figure 5.1c return the initial value of x satisfies PC (see section 5.4.1 for a formal definition). Instead, we need a litmus program like in Figure 5.1d which contains two FAAs per session. Here, an execution where all FAAs return an initial value does not satisfy PC. This program can also be used to show the non-existence of available implementations of *Parallel Snapshot Isolation* (PSI) [98] or *Conflict-preserving Causal Consistency* (CCC), a consistency model defined using the axioms Conflict and Causal from [29]. As a side remark, note that CCC is equivalent to CC for the key-value store with PUT and GET operations presented at the beginning, and therefore, there exists an available implementation for CCC in that case.

While these cases follow a broadly similar proof strategy, each demands distinct proof artifacts (such as litmus programs) and tailored reasoning. The AFC theorem unifies these diverse arguments within a common theoretical framework, grounded in a formalization of a wide class of storage specifications encompassing all the examples above.

5.3 Abstracting Storage Executions

We present an abstract model of distributed storage executions that includes the essential components needed to define storage specifications. A *distributed storage* (or simply storage) replicates the state of a set of objects over two or more nodes called *replicas*. We use Keys to denote the infinite set of objects, ranged over x, y, z, and Reps to denote the set of replica identifiers, ranged over r, r_1, r_2. Objects are accessed using a set of *operations* which may write or return values in a set Vals.

An *abstract execution* records operation invocations along with a set of relations that represent control-flow dependencies (two invocations executing on the same replica), and the internal behavior of the storage. The internal behavior includes, broadly, the computation of object states and the return values of invocations, as well as the communication between replicas. The first concerns local computation within each replica, while the second pertains to communication protocols or underlying network assumptions. To distinguish these two aspects, we first introduce the concept of a *history*, which records only the data-flow dependencies relevant to characterizing the local computation. An abstract execution is then defined as an extension of a history, enriched with additional relations that abstract inter-replica communication.

5.3.1 Histories

The invocation of an operation on some replica is represented using an *event* $e = (\mathsf{id}, \mathsf{r}, \mathsf{op}, \mathsf{wval}, \mathsf{m})$ where id is an event identifier, r is a replica identifier, op is an operation name, wval is a (partial) mapping that associates an object x with a value v that this event writes to x, and m is additional metadata of the invocation. We use $\mathtt{id}(e)$ $\mathtt{rep}(e), \mathtt{op}(e)$, $\mathtt{wval}(e)$, and $\mathtt{md}(e)$ to denote the event identifier, replica identifier, operation, written value mapping and metadata of an event e, respectively. We assume that every event e accesses (reads or writes) a fixed finite set of objects denoted as $\mathsf{obj}(e)$. The set of events is denoted by Events. We assume that Events includes a distinguished type of *initial events* that affect every object, representing the initial state of the storage.

Example 5.3.1. *As a running example, we consider a Key-value Store with four types of operations:* $\mathtt{PUT}(x, v)$ *that writes* v *to object (key)* x*,* $\mathtt{GET}(x)$ *that reads object* x*,* $\mathtt{FAA}(x, v)$ *that reads the value* v' *of object* x *and writes* $v' + v$*, and* $\mathtt{CAS}(x, v, v')$*, that reads* x *and writes* v' *iff the value read is* v*. We use* faacas *to refer to this storage (from the Fetch-and-Add and Compare-and-Swap operations). Section 5.9.1 summarizes the full description of* faacas.

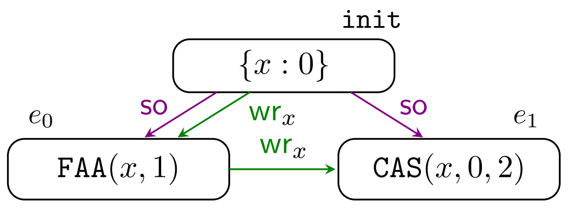
We generalize the notion of history presented in Chapter 2 for storage specifications.

A *history* contains a finite set of events E ordered by a (partial) *session order* so that relates events on the same replica, and a *write-read* relation wr (also known as read-from) representing data-flow dependencies between events that update and respectively, read a same object. Histories contain an initial event, $\mathtt{init}$, that precedes every other event in E w.r.t so. We consider a write-read relation $\mathsf{wr}_x \subseteq \mathcal{P}(E) \times E$ for every object $x \in \mathsf{Keys}$. The inverse of wr_x is defined as usual and denoted by wr_x^{-1}. We use $\mathsf{wr} : \mathsf{Keys} \to \mathcal{P}(E) \times E$ to denote the mapping associating each object x with wr_x.

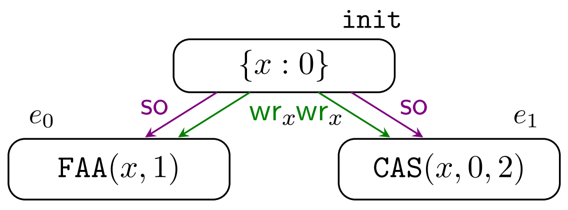
For simplicity, we often abuse the notation and extend wr_x and wr to pairs of events: we say that $(w, r) \in \mathsf{wr}_x$ if $w \in \mathsf{wr}_x^{-1}(r)$, and we say that $(w, r) \in \mathsf{wr}$ if there exists an object x s.t. $(w, r) \in \mathsf{wr}_x$.

Definition 5.3.2. *A* history $(E, \mathsf{so}, \mathsf{wr})$ *is a finite set of events* E *along with a strict partial* session order so*, and a* write-read *relation* $\mathsf{wr}_x \subseteq \mathcal{P}(E) \times E$ *for every* $x \in \mathsf{Keys}$ *such that*

- E *contains a single initial event* $\mathtt{init}$*, which precedes every other event in* E *w.r.t.* so*,*
- $\forall e, e' \in E \setminus \{\mathtt{init}\}$*,* so *orders* e *and* e' *iff* $\mathtt{rep}(e) = \mathtt{rep}(e')$*,*
- *the inverse of* wr_x *is a total function for every* $x \in \mathsf{Keys}$*, and*
- $\mathsf{so} \cup \mathsf{wr}$ *is acyclic (here we use the extension of* wr *to pairs of events).*



(a) e_0 reads 0, writes 1; e_1 reads 1 and does not write.



(b) e_0 reads 0 and writes 1; e_1 reads 0 and writes 2.

Figure 5.2: Two examples of histories for faacas. Arrows represent so and wr relations. The initial event `init` defines the initial state where x is 0. Events e_0 and e_1 execute a fetch-and-add and compare-and-swap respectively, at different replicas.

Example 5.3.3. *Figure 5.2 shows two examples of histories of the storage* faacas *presented in Example 5.3.1. For readability, we omit replica identifiers from events. The* wr *dependencies can be used to explain the "local" computation in those invocations as follows: (1) on the left, the* `CAS` *should fail (not write to x) because it reads the value written by the* `FAA` *which should be equal to 1 since* `FAA` *reads the initial value, (2) on the right, the* `CAS` *should succeed (write to x) because it reads the initial value (the* `FAA` *will concurrently write 1 to x).*

We say that the event w is *read* by the event r if $(w, r) \in \mathsf{wr}$. Since we assumed that wr_x^{-1} is a total function, we use $\mathsf{wr}_x^{-1}(r)$ to denote the set W such that $(\mathsf{W}, r) \in \mathsf{wr}_x$. We use $\mathsf{wr}_x^{-1}(e) = \emptyset$ to indicate that e does not read x (resp. $\mathsf{wr}_x^{-1}(e) \neq \emptyset$ to indicate that e reads x).

5.3.2 Abstract Executions

An *abstract execution* of a distributed storage is a history with a finite set of events E along with a relation $\mathsf{rb} \subseteq E \times E$ called *receive-before*, and a total order $\mathsf{ar} \subseteq E \times E$ called *arbitration*. These relations are an abstraction of the internal communication behavior, i.e., the propagation of operation invocations between different replicas and conflict-resolution policies. The receive-before relation models information exchange between replicas and intuitively, an event w is received-before an event e on a replica r if w has been propagated to replica r before executing e. The arbitration order represents a "last-writer wins" conflict resolution policy between concurrent events and the order in which events take effect in the storage for "strong" consistency models such as Sequential Consistency or Serializability. This order may be ignored by weaker consistency models, where a read is *not* required to read from the latest update that precedes it in arbitration order, or by specific types of storage, e.g., CRDTs (see Section 5.8), where conflict resolution does not rely on the arbitration order.

Definition 5.3.4. *An* abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ *is a history* $h = (E, \mathsf{so}, \mathsf{wr})$ *along with an asymmetric, irreflexive relation* receive-before $\mathsf{rb} \subseteq E \times E$ *and a strict total* arbitration order $\mathsf{ar} \subseteq E \times E$*, such that:*

1. *propagated updates are not "forgotten" within the same replica:* $\mathsf{rb} = \mathsf{rb}; \mathsf{so}^*$[2],

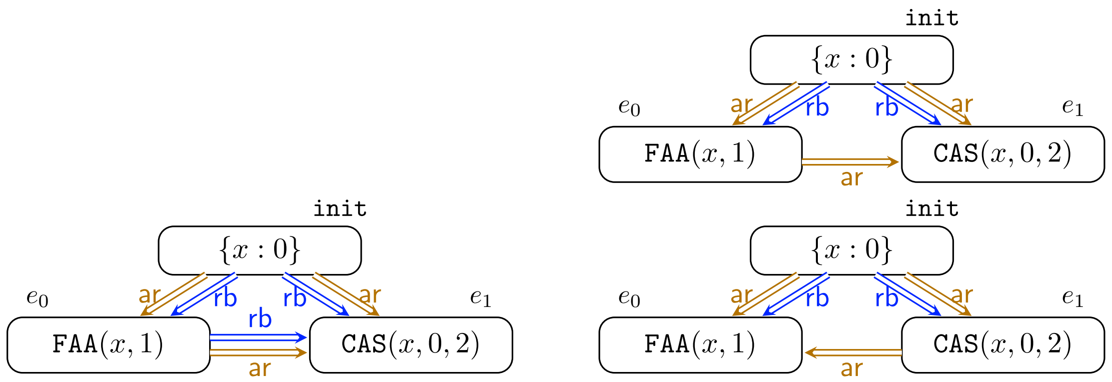
[2]The symbol ; denotes the usual composition of relations

2. *events at the same replica or events that are read are necessarily received-before, and* ar *is consistent with the receive-before relation:* $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{rb} \subseteq \mathsf{ar}$.

ξ is called an abstract execution of h.

The conditions above are naturally satisfied by storages where replicas execute in a single process, values are not produced "out of thin air", and the arbitration order is implemented using "consistent" timestamps, i.e. timestamps that do not contradict Lamport's clocks [76] or causality. This is the case for implementations where "ties" between concurrent operations are solved based on replica IDs (assumed to be totally ordered), or when using timestamps from a (partially-)synchronized clock – which is most often the case in practice.

For an event e, we use $e \in \xi$ to denote the fact that $e \in E$.



(a) Abstract execution of the history in Figure 5.2a.

(b) Two abstract executions of the history in Figure 5.2b.

Figure 5.3: Abstract executions of the histories from Figure 5.2. Arrows represent ar and rb relations. For readability, we omit the so and wr relations. The event e_0 is received-before executing e_1 in Figure 5.3a but not in Figure 5.3b. The arbitration relation is the same in both executions.

Example 5.3.5. *Figure 5.3 shows abstract executions for the histories in Figure 5.2. In both cases, the receive-before relation includes only the* wr *dependencies which is anyway required by definition. Reading a value at some replica r produced by an invocation e at some other replica r' should imply that e propagated to r. On the left, the arbitration order includes just the* wr *dependencies which already ensure totality. On the right,* `FAA` *and* `CAS` *are concurrent, i.e., both invocations were executed before either had a chance to propagate. We present the two possible arbitration orders. This shows that the arbitration order cannot be always determined based on the information exchanged between the replicas, i.e. by the receive-before.*

The concept of abstract execution defined earlier is subsequently used to formalize the specifications of distributed storage systems. We will start with a so-called basic class that concerns "single-object" operations.

5.4 Basic Storage Specifications

We present a first class of storage specifications, called *basic*, where operations read and/or write a *single value* from/to a *single object* (the operations in Example 5.3.1 satisfy this assumption). We will present a more general framework with multi-object operations that read and/or write multiple values or objects in Section 5.8.

In general, a storage specification has two parts: a *consistency model* characterizing the propagation of invocations between different replicas, and an *operation specification* which defines object states and return values. The definition of consistency models generalizes the definition of isolation level presented in Chapter 2, and the definition of operation specifications refines replicated data types as defined in [40]. The first two subsections define these concepts for the class of operations mentioned above, and the last subsection formalizes the validity of an abstract execution w.r.t. such storage specifications.

5.4.1 Basic Consistency Models

In general, a consistency model is defined as a non-empty set of *visibility* formulas that characterize the *context* in which an event (invocation of an operation) is executed (abstractly speaking). The context of an event e at a replica r is defined as the set of events, potentially from other replicas, that propagated to r prior to executing e. The notion of validity w.r.t. a consistency model defined later will require that the event e which is read by another event e' is the last in the arbitration order ar within the context of e'. This accurately models the Last-Writer-Wins conflict resolution policy (we consider other conflict resolution policies in Section 5.8). We define hereafter a class of so-called basic consistency models that will be extended later in Section 5.8.1.

Formally, a visibility formula v describes a binary relation between events which is parametrized by an object in Keys. This is written as a predicate $\mathsf{v}_x(e_1, e_2)$ meaning that v relates e_1 to e_2 for object x (explained below). A *consistency model* (criterion) CMod is a set of visibility formulas.

For a consistency model CMod and an abstract execution ξ, the context of an event r for object x is the set of all events e which are related to r by some visibility formula in CMod along with a projection of rb and ar to this set of events, i.e.,

$$\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}]) = (E_x, \mathsf{rb}_{E_x \times E_x}, \mathsf{ar}_{E_x \times E_x}) \text{ with } E_x = \{e \in \xi \mid \exists \mathsf{v} \in \mathsf{CMod}.\ \mathsf{v}_x(e, r)\} \quad (5.1)$$

We use $\mathsf{Contexts}$ to denote the set of all possible contexts, i.e., tuples $(E, \mathsf{rb}_E, \mathsf{ar}_E)$ where E is a finite set of events, rb_E is an asymmetric, irreflexive relation over E, and ar_E is a strict total order over E, such that $\mathsf{rb}_E \subseteq \mathsf{ar}_E$.

Basic visibility formulas (used in basic consistency models) have the following form:

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_n) ::= \exists \varepsilon_1, \ldots, \varepsilon_{n-1}.\ \bigwedge_{i=1}^{n} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^{\mathsf{v}} \ \wedge \varepsilon_0 \ \mathsf{writes}\ x \wedge \mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset \quad (5.2)$$

where each relation $\mathsf{Rel}_i^{\mathsf{v}}$, $1 \leq i \leq n$, is defined by the grammar listed below:

$$\mathsf{Rel} ::= \mathtt{id} \mid \mathsf{so} \mid \mathsf{wr} \mid \mathsf{rb} \mid \mathsf{ar} \mid \mathsf{Rel} \cup \mathsf{Rel} \mid \mathsf{Rel}; \mathsf{Rel} \mid \mathsf{Rel}^? \mid \mathsf{Rel}^+ \mid \mathsf{Rel}^* \quad (5.3)$$

This formula states that ε_0 (which is e in Eq.5.1) is connected to ε_n (which is r in Eq.5.1) by a path of dependencies that go through some intermediate events $\varepsilon_1, \ldots \varepsilon_{n-1}$ (all the ε variables are interpreted as events). The constraint $\mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset$ asks that ε_n reads the object x. Every relation used in the path is a composition of $\mathsf{so}, \mathsf{wr}, \mathsf{rb}$ and ar via union $\cup$, composition of relations ;, and transitive closure $^+$. $\mathsf{Rel}^?$ is syntactic sugar for $\mathtt{id} \cup \mathsf{Rel}$, and Rel^* for $\mathtt{id} \cup \mathsf{Rel}^+$. Since the grammar includes composition the existential quantifiers in Eq.5.3 do not increase expressivity (one could write $(\varepsilon_0, \varepsilon_{n+1}) \in \mathsf{Rel}_1^{\mathsf{v}}; \ldots; \mathsf{Rel}_n^{\mathsf{v}}$). These quantifiers are used to simplify proofs in Section 5.6.

The predicate ε_0 writes x means that ε_0 writes to object x, i.e., $\mathtt{wval}(e)(x) \downarrow$.

We write $\mathsf{v}_x(e_0, \ldots e_n)$ whenever $\mathsf{v}_x(e_0, e_n)$ holds using the events $e_1, \ldots e_{n-1}$ to instantiate the existential quantifiers. The length of v_x, denoted by $\mathsf{len}(\mathsf{v}_x)$, is the number of relations $\mathsf{Rel}_i^{\mathsf{v}}$ used in its definition (n in Equation (5.2)).

$$\begin{tikzcd}
\varepsilon_0 \arrow[d, dashed, Rightarrow, "\mathsf{ar}"'] \arrow[dr, bend left, "\mathsf{so} \cup \mathsf{wr}"] & \\
e \arrow[r, "\mathsf{wr}_x"] & \varepsilon_1
\end{tikzcd}$$

writes x

ε_0 writes x $\wedge$ $\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge$
$(\varepsilon_0, \varepsilon_1) \in \mathsf{so} \cup \mathsf{wr}$

(a) Return-Value

$$\begin{tikzcd}
\varepsilon_0 \arrow[d, dashed, Rightarrow, "\mathsf{ar}"'] \arrow[dr, bend left, Rightarrow, "\mathsf{rb}^+"] & \\
e \arrow[r, "\mathsf{wr}_x"] & \varepsilon_1
\end{tikzcd}$$

writes x

ε_0 writes x $\wedge$ $\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge$
$(\varepsilon_0, \varepsilon_1) \in \mathsf{rb}^+$

(b) Causal

$$\begin{tikzcd}
\varepsilon_0 \arrow[d, dashed, Rightarrow, "\mathsf{ar}"'] \arrow[r, Rightarrow, "\mathsf{ar}^*"] & \bullet \arrow[d, "(\mathsf{so} \cup \mathsf{wr})"'] \\
e \arrow[r, "\mathsf{wr}_x"] & \varepsilon_1
\end{tikzcd}$$

writes x

ε_0 writes x $\wedge$ $\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge$
$(\varepsilon_0, \varepsilon_1) \in \mathsf{ar}^*; (\mathsf{so} \cup \mathsf{wr})$

(c) Prefix

$$\begin{tikzcd}
\varepsilon_0 \arrow[d, dashed, Rightarrow, "\mathsf{ar}"'] \arrow[dr, bend left, Rightarrow, "\mathsf{ar}"] & \\
e \arrow[r, "\mathsf{wr}_x"] & \varepsilon_1
\end{tikzcd}$$

writes x

ε_0 writes x $\wedge$ $\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \wedge$
$(\varepsilon_0, \varepsilon_1) \in \mathsf{ar}$

(d) SC / SER

Figure 5.4: Visibility formulas defining the homonymous consistency models *Return-Value Consistency* (RVC, Figure 5.4a), *Causal Consistency* (CC, Figure 5.4b), *Prefix Consistency* (PC, Figure 5.4c) and *Sequential Consistency/Serializability* (SC/SER, Figure 5.4d). Solid edges describe the dependencies linking ε_0 and ε_1. We include the wr_x edge (and its source e) as a visualization of the constraint $\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset$. Dashed ar edges are not part of the visibility formulas. These capture the Last-Writer-Wins conflict resolution policy discussed later, requiring that the event e being read succeeds all other events from the context in ar.

As mentioned above, a *basic consistency model* is a set of basic visibility formulas.

Figure 5.4 describes several visibility formulas and their corresponding consistency models, inspired by Biswas et al. [29]. The dashed ar edges (leading to e) should be ignored for now.

Basic visibility formulas constrain events w.r.t. a single object – x. Later, we will define consistency models whose visibility formulas can impose additional constraints that concern multiple objects.

We say that a consistency model CMod_1 is *weaker than* another consistency model CMod_2, denoted $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$ if intuitively, the context of any event w.r.t. CMod_1 is larger than the context w.r.t. CMod_2. Formally, $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$ iff for every abstract execution ξ, event $e \in \xi$ and object x, $\mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}_2])$ holds. CMod_1 and CMod_2 are *equivalent*, denoted $\mathsf{CMod}_1 \equiv \mathsf{CMod}_2$, when $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$ and $\mathsf{CMod}_2 \preccurlyeq \mathsf{CMod}_1$.

We assume that every consistency model CMod includes a visibility formula $\mathsf{v}_x^{\mathsf{so}}$ (resp. $\mathsf{v}_x^{\mathsf{wr}}$) such that $\mathsf{so} \subseteq \mathsf{v}_x^{\mathsf{so}}$ (resp. $\mathsf{wr}_x \subseteq \mathsf{v}_x^{\mathsf{wr}}$) for every object $x \in \mathsf{Keys}$. The constraint $\mathsf{so} \subseteq \mathsf{v}_x^{\mathsf{so}}$ corresponds to the so-called "read-my-own-writes" consistency (i.e., an event "observes" every preceding event at the same replica) and $\mathsf{wr}_x \subseteq \mathsf{v}_x^{\mathsf{wr}}$ is a “well-formedness” constraint since visibility formulas will constrain the write-read relation in a history (see Definition 5.4.2).

All consistency models in Figure 5.4 trivially satisfy this constraint as for any abstract execution, $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{rb} \subseteq \mathsf{ar}$. `RVC` is the weakest consistency model that our framework can describe.

5.4.2 Basic Operation Specifications

While visibility formulas define the context of an invocation in terms of prior invocations, the effect of an invocation is defined using the following semantical functions: rspec says whether an event reads an object or not, and wspec defines the value written by the invocation, if any. The written value may depend on the value read by the event in the case of atomic read writes like `FAA` and `CAS`. Concerning notations, for a partial function $f : A \rightharpoonup B$, we use $f(a) \downarrow$ to say that f is defined for $a \in A$, and $f(a) \uparrow$, otherwise. Similarly, for a predicate p over some set A, we use $p(a) \downarrow$ to say that p is true for a, and $p(a) \uparrow$, otherwise.

A *basic read specification* rspec is a predicate over Events. For example, Equation (5.4) describes the read specification of faacas. We say that an event e is a *read* event if $\mathsf{rspec}(e) \downarrow$, and in such case, we say that e reads $\mathsf{obj}(e)$.

$$\mathsf{rspec}(r) = \mathsf{true} \text{ iff } \mathsf{op}(r) = \mathtt{GET}, \mathtt{FAA}, \mathtt{CAS} \tag{5.4}$$

A *basic write specification* wspec is a partial function $\mathsf{wspec} : \mathsf{Events} \rightharpoonup \mathsf{Vals} \rightharpoonup \mathsf{Vals}$, that associates non-initial events to partial functions that map a read value to a value to be written. For example, Equation (5.5) describes the write specification of faacas.

$$\mathsf{wspec}(w)(v) = \begin{cases} v' & \text{if } w = \mathtt{PUT}(x, v') \\ v + v' & \text{if } w = \mathtt{FAA}(x, v') \\ v'' & \text{if } w = \mathtt{CAS}(x, v', v'') \land v = v' \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.5}$$

For an event e, we say that e is a *write* event if $\mathsf{wspec}(e) \downarrow$. We assume that if $\mathsf{wspec}(e) \downarrow$, then the function $\mathsf{wspec}(e) : \mathsf{Vals} \rightharpoonup \mathsf{Vals}$ is defined for at least one value. We say that e writes x given v if $x = \mathsf{obj}(e)$ and $\mathsf{wspec}(e)(v) \downarrow$. We assume that every value v can *enable* at least one event to write, i.e., there exists $e \in \mathsf{Events}$ s.t. $\mathsf{wspec}(e)(v) \downarrow$. We also assume that if e is a write event but it is not a read event, e.g., a `PUT` invocation, then $\mathsf{wspec}(e)$ is a total constant function, i.e. $\mathsf{wspec}(e) : \mathsf{Vals} \to \mathsf{Vals}$ and $\mathsf{wspec}(e)(v_1) = \mathsf{wspec}(e)(v_2)$ for all v_1, v_2.

Definition 5.4.1. *A* basic operation specification *is a tuple* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{wspec})$ *where E is a set of events, such that* $\mathsf{obj}(e)$ *is a singleton for every $e \in E$.*

We use $\mathsf{Events}[\mathsf{OpSpec}]$ to refer to the set of events E in an operation specification.

Operation Closure. We define some natural assumptions about basic operation specifications (it is easy to check that they hold on the faacas example with the definitions in Equation (5.4) and Equation (5.5)). We assume that E contains at least one read and one write event. We also assume that all objects support a common set of operations with identical read and write behavior, and that these operations can be executed at any replica. Formally, for every event $e \in E$, replica r, identifier id and object x there exists an event $e' \in E$ s.t. $\mathtt{rep}(e') = \mathsf{r}$, $\mathsf{id}(e') = \mathsf{id}$, $\mathsf{obj}(e') = x$, $\mathsf{rspec}(e') = \mathsf{rspec}(e)$ and $\mathsf{wspec}(e') = \mathsf{wspec}(e)$.

(Conditional) Read-Write Events. We say that OpSpec allows read-writes if E contains an event that is a read and a write event at the same time (e.g., `FAA` and `CAS` invocations); we call such events *read-write* events. If OpSpec allows read-writes, then we assume that every value can *enable* some read-write to write, i.e., for every value v, E contains a read-write event e s.t. $\mathsf{wspec}(e)(v) \downarrow$. As an example, this condition is not satisfied by a storage with only `GET` and `TEST&SET` operations (`TEST&SET` writes 1 if it reads 0 and nothing otherwise). Indeed, value 1 cannot enable any write.

A read-write event is called *unconditional* if for every value v, $\mathsf{wspec}(e)(v) \downarrow$ and *conditional* otherwise. For example, a `FAA` invocation is unconditional and a `CAS` invocation is conditional. We assume that if OpSpec allows conditional writes, then every value v can *disable* some conditional read-write to write, i.e., E contains a conditional read-write event s.t. $\mathsf{wspec}(e)(v) \uparrow$.

5.4.3 Validity w.r.t. Basic Storage Specifications

A *basic storage specification* is a pair $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ where CMod is a basic consistency model and OpSpec is a basic operation specification. Next, we formalize the validity of an abstract execution w.r.t. a basic storage specification.

The interpretation of a basic visibility formula $\mathsf{v}_x(\varepsilon_0, \varepsilon_n)$ on an abstract execution ξ is defined as expected.

Definition 5.4.2. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a basic storage specification. An abstract execution* $\xi = (h, \mathsf{rb}, \mathsf{ar})$ *of a history* $h = (E, \mathsf{so}, \mathsf{wr})$ *is* valid *w.r.t.* Spec *iff*

- *it contains events from the operation specification, i.e.,* $E \subseteq \mathsf{Events}[\mathsf{OpSpec}]$*,*
- *the write-read dependencies of each event $e \in E$ for object x satisfy the following:*
  - *if e reads object x, i.e.* $\mathsf{rspec}(e) \downarrow$ *and* $x \in \mathsf{obj}(e)$*, e reads from the write event in its context that is maximal w.r.t. the arbitration order:* $\mathsf{wr}_x^{-1}(e) = \{w_x^e\}$*,*
  - *if e does not read object x, i.e.* $\mathsf{rspec}(e) \uparrow$ *or* $x \notin \mathsf{obj}(e)$*, then* $\mathsf{wr}_x^{-1}(e) = \emptyset$*.*
- *the value written by each event $e \in E$ to object x is consistent with* wspec*:*

– *if e reads object x, i.e.* $\mathsf{rspec}(e)\downarrow$ *and* $x \in \mathsf{obj}(e)$*, then it writes based on the value read:* $\mathtt{wval}(e)(x) = \mathsf{wspec}(e)(\mathtt{wval}(w^e_x)(x))$[3]*,*

– *if e does not read object x, i.e.* $\mathsf{rspec}(e)\uparrow$ *or* $x \notin \mathsf{obj}(e)$*, then* $\mathtt{wval}(e)(x) = \mathsf{wspec}(e)(\_)$[4]*,*

where $w^e_x = \max_{\mathsf{ar}} \mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}])$.

A history h is valid w.r.t. Spec *iff there exists an abstract execution of h which is valid w.r.t.* Spec.

Recall that the `value` function, and implicitly, the operation specification, are used to interpret the visibility formulas of CMod and thus define invocation contexts.

Example 5.4.3. *The abstract executions described in Figure 5.3 are both valid w.r.t.* $(\mathsf{CC}, \mathsf{faacas})$ *as every event which is read is also received-before* ($\mathsf{wr} \subseteq \mathsf{rb}$)*. However, only Figure 5.3a is valid w.r.t.* $(\mathsf{SC}, \mathsf{faacas})$*. In Figure 5.3a,* e_1 *reads from the writing event that precedes it w.r.t.* ar*. On the other hand, in Figure 5.3b,* e_1 *reads x from* `init` *and not from* e_0 *which is its maximal visible event w.r.t.* ar *that writes x. Moreover, by the symmetry between* e_0 *and* e_1*, it can be proven that any abstract execution of such history is not valid w.r.t.* $(\mathsf{SC}, \mathsf{faacas})$*.*

5.5 Programs and Storage Implementations

We model programs accessing a storage and storage implementations using *Labeled Transition Systems (LTSs)*. Their interaction via invocations of operations will be defined as the usual parallel composition of LTSs. We also present the notions of availability and validity of a storage implementation, key to the AFC theorem.

5.5.1 Labeled Transition Systems

An LTS $L = (S, A, s_0, \Delta)$ is a tuple formed of a (possibly infinite) set of *states* S, a set of *actions* A, an *initial state* $s_0 \in S$ and a (partial) *transition function* $\Delta : S \times A \rightharpoonup S$. An *execution* of L is an alternating sequence of states and actions $\rho = s_0, a_0, s_1, a_1, s_2, \ldots$ such that $\Delta(s_i, a_i) = s_{i+1}$ for each i. A state s is *reachable* if there exists an execution ending in s. A *trace* of an execution ρ is the projection of ρ over actions (the maximum subsequence of ρ formed of actions). The final state of a finite trace t, denoted by $\mathtt{state}(t)$, is the last state of ρ. The set of all traces of L is denoted by $\mathcal{T}_L$. An LTS is *finite* if all its traces are finite. For any finite trace t and action a, $\Delta(t, a)$ is defined as $\Delta(\mathtt{state}(t), a)$. If $\Delta(t, a)\downarrow$, then $t \oplus a$ is defined by appending a to t.

Let $L_1 = (S_1, A_1, s^1_0, \Delta_1)$ and $L_2 = (S_2, A_2, s^2_0, \Delta_2)$ be two LTSs. We define a parallel composition operator between L_1 and L_2 that is parametrized by a partial function $\pi : A_1 \rightharpoonup A_2$. This function allow us to define a relationship between a subset of A_1 and a subset of A_2, called *synchronized actions* of L_1 and L_2. The set of actions $a \in A_1$ for which $\pi(a)$ is not defined (resp. actions $a \in A_2$ for which $\pi^{-1}(a)$ is not defined) are the *local actions* of L_1

[3]Since `wval` and `wspec` are partial functions, the equality also means that the left side is defined iff the right side is defined.

[4]_ represents any value in Vals.

(resp. L_2). Without loss of generality, we assume that the set of local actions of L_1 and L_2 are disjoint.

The parallel composition of L_1 and L_2 w.r.t. π is the LTS $L_1 \parallel_\pi L_2 = (S, A, s_0, \Delta)$ where $S = S_1 \times S_2$, $A = A_1 \cup A_2$, $s_0 = (s_0^1, s_0^2)$, and Δ is defined as follows:

$$\Delta((s_1, s_2), a) ::= \begin{cases} (\Delta(s_1, a), \Delta(s_2, \pi(a))) & \text{if } a \in A_1, \pi(a) \downarrow, \Delta(s_1, a) \downarrow, \text{ and } \Delta(s_2, \pi(a)) \downarrow \\ (\Delta(s_1, a), s_2) & \text{if } a \in A_1, \pi(a) \uparrow, \text{ and } \Delta(s_1, a) \downarrow \\ (s_1, \Delta(s_2, a)) & \text{if } a \in A_2, \pi^{-1}(a) \uparrow, \text{ and } \Delta(s_2, a) \downarrow \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

(note the asymmetry due to using the function π). Whenever there is no ambiguity w.r.t. π we simply write $L_1 \parallel L_2$.

5.5.2 Programs and Storage Implementations

Let E be a set of events. A *program* over E is an LTS $P_E = (S_\mathsf{p}, A_\mathsf{p}, s_0^\mathsf{p}, \Delta_\mathsf{p})$ such that $E \subseteq A_\mathsf{p}$. Intuitively, this LTS models all possible interleavings between invocations on different replicas. Actions in $A_\mathsf{p} \setminus E$ represent computation steps performed by the program locally, before or after invoking operations on the storage. Also, to simplify the technical exposition, we do not consider separate transitions for calling and returning from a storage operation. Intuitively, the transitions labeled by events occur at the return time.

A *storage implementation* over E is an LTS $I_E = (S_\mathsf{i}, A_\mathsf{i}, s_0^\mathsf{i}, \Delta_\mathsf{i})$ such that A_i contains (1) an arbitrary set of local actions (representing computation/communication steps internal to the storage), and (2) pairs of events in E and their read-dependencies, i.e., pairs (e, m) where $e \in E$ and $m : \mathsf{Keys} \rightharpoonup \mathcal{P}(E)$. Intuitively, m represents the write-read dependencies of e. We also assume that each action includes an identifier, denoted by $\mathtt{id}(a)$, so that along an execution every action occurs only once. For any action $a = (e, m)$, $\mathtt{ev}(a)$ and $\mathsf{wr}\text{-}\mathtt{Set}(a)$ denote the event e and the write-read dependencies m respectively. Also, $\mathtt{op}(a) = \mathtt{op}(e)$ is the operation type of a. To model communication, we assume that A_i includes two types of local actions, `send` actions for sending a message (from one replica to another) and `receive` to receive a message.

The formalization of send/receive actions is straightforward and we omit it. We will say that a send action *matches* a receive action if they concern precisely the same message (messages are associated with unique identifiers). For any `send`, resp., `receive`, action a at some replica r, $\mathsf{rb}\text{-}\mathtt{Set}(a)$ denotes the set of events that r sends in this message, resp., that r receives in this message. We assume that if a trace t contains any such action, for every event $e \in \mathsf{rb}\text{-}\mathtt{Set}(a)$ there must exist an action $(e, \_)$ preceding a in t. As expected, if a_s and a_r match, then $\mathsf{rb}\text{-}\mathtt{Set}(a_s) = \mathsf{rb}\text{-}\mathtt{Set}(a_r)$.

For any action $a \in A_\mathsf{p} \cup A_\mathsf{i}$, $\mathtt{rep}(a)$ denotes the replica executing a.

The interaction between a storage implementation I_E and a program P_E is defined as their asymmetric parallel composition w.r.t. a partial function $\pi : A_\mathsf{i} \rightharpoonup A_\mathsf{p}$ which is defined only for actions of the form (e, m) (as described above) by $\pi(e, m) = e$. The program and the storage implementation synchronize on events representing operation invocations. It is denoted by $I_E \parallel P_E$. By definition, traces of $I_E \parallel P_E$ include actions of the form (e, m) (coming from A_i), and local actions of P_E or I_E.

Traces of I_E (or $I_E \parallel P_E$) induce histories and abstract executions. The *induced history* of a trace t of I_E (or $I_E \parallel P_E$) is the history $h = (E^t, \mathsf{so}^t, \mathsf{wr}^t)$ where E^t is the set events e such that some action $a_e = (e, m)$ occurs in t, so^t orders events from the same replica as they occur in t, and for every object x and event e, $(\mathsf{wr}^t_x)^{-1}(e) = \mathsf{W}$ iff $\mathsf{wr}\text{-}\mathtt{Set}(a_e) = (x, \mathsf{W})$ (a_e is the action that contains e). We implicitly assume that for any event $e \in E$ different from $\mathtt{init}$, $(\mathtt{init}, e) \in \mathsf{so}^t$. We use $\mathsf{h}(t)$ to denote the induced history of a trace t.

The *induced receive-before* of a trace t of I_E (or $I_E \parallel P_E$) is the relation rb^t over events induced by the matching relation between sends and receives: $(e, e') \in \mathsf{rb}^t$ iff $(e, e') \in \mathsf{so}^t$ or there exists matching $\mathtt{send}$ and $\mathtt{receive}$ actions, a_s, a_r and a synchronized action $a = (e', \_)$ s.t. $\mathtt{rep}(a_r) = \mathtt{rep}(a)$, a_r occurs before a in t, and $e \in \mathsf{rb}\text{-}\mathtt{Set}(a_s)$ (which coincides with $\mathsf{rb}\text{-}\mathtt{Set}(a_r)$).

A trace t of I_E also induces a set of abstract executions of the form $\xi = (\mathsf{h}(t), \mathsf{rb}^t, \mathsf{ar}^t)$ where ar^t is any total order between the events in ξ that is consistent with rb^t, i.e., $\mathsf{rb}^t \subseteq \mathsf{ar}^t$ (to satisfy the requirements in Definition 5.3.4).

5.5.3 Availability and Validity of a Storage Implementation

We say that a storage implementation I_E is *available* if, intuitively, every execution of I_E terminates when interacting with a finite program P_E (executing a single synchronized action does not make a replica enter an infinite loop of local steps), and no invocation is delayed due to a replica waiting for messages.

We say that a replica $r \in \mathsf{Reps}$ is *waiting* in a trace t of some composition $I_E \parallel P_E$ if

- the program can execute some action at replica r: there is an action $a \in A_{\mathsf{p}}$ s.t. $\mathtt{rep}(a) = r$ and $\Delta_{P_E}(t', a) \downarrow$; where t' is obtained from t by removing all local actions of I_E and replacing every action (e', m) with e', and
- the only actions of replica r that the parallel composition can execute are $\mathtt{receive}$ actions: for every action $a \in A_{\mathsf{p}} \cup A_{\mathsf{i}}$ s.t. a is not a $\mathtt{receive}$ action and $\mathtt{rep}(a) = r$, $\Delta_{I_E \parallel P_E}(t, a) \uparrow$.

Note that the latter implies that the action a that P_E can execute after t' is necessarily an event in E (otherwise, a is a local action of P_E and the parallel composition could execute it).

Definition 5.5.1. *An implementation I_E is* available *if the following hold:*

- *for every finite program P_E, the composition $I_E \parallel P_E$ is also finite, and*
- *for every program P_E and every trace t of $I_E \parallel P_E$, there is no replica waiting in t.*

Given a storage specification Spec over a set of events E, a storage implementation I_E is *valid w.r.t.* Spec if every trace t induces some abstract execution which is valid w.r.t. Spec.An implementation valid w.r.t. Spec is simply called a Spec-*implementation* (or implementation of Spec).

5.6 The Basic Arbitration-Free Consistency Theorem

We present a simpler instance of our main result (the AFC theorem) for basic storage specifications.

To simplify the statement of the theorem, we define a normal form for basic consistency models w.r.t. a basic operation specification OpSpec. A visibility formula is called *simple* if it does *not* use composition operators between relations, i.e., the grammar in Equation (5.3) is replaced by: $\mathsf{Rel} ::= \ \mathsf{so} \mid \mathsf{wr} \mid \mathsf{rb} \mid \mathsf{ar}$. Also, a visibility formula v from a consistency model CMod is called *vacuous* w.r.t. OpSpec iff for every abstract execution ξ, ξ is valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$ iff ξ is valid w.r.t. $(\mathsf{CMod} \setminus \{\mathsf{v}\}, \mathsf{OpSpec})$. For example, if $\mathsf{Rel}_i^{\mathsf{v}}$ and $\mathsf{Rel}_{i+1}^{\mathsf{v}}$ in Equation (5.2) are wr (for some i), then any instance of ε_i must be an invocation of a read-write that both reads and writes. If the operation specification does not include read-writes (e.g., a key-value store with only `PUT` and `GET` operations), such visibility formulas are vacuous.

Definition 5.6.1. *A basic consistency model* CMod *is called in* normal form w.r.t. a basic operation specification OpSpec *if it contains only simple visibility formulas and no visibility formula from* CMod *is vacuous w.r.t.* OpSpec.

A normal form of a basic consistency model CMod w.r.t. OpSpec is any basic consistency model CMod' in normal form, such that for every abstract execution ξ, ξ is valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$ iff ξ is valid w.r.t. $(\mathsf{CMod}', \mathsf{OpSpec})$. We show in Section 5.11 that every basic consistency model CMod has a normal form. A normal form can be obtained by replacing each visibility formula v with an equivalent (possibly infinite) set of simple visibility formulas S_{v}. Each set S_{v} is obtained by recursively decomposing the union, composition and transitive closure operators in each relation $\mathsf{Rel}^{\mathsf{v}}$ (see Equation (5.2)).

A visibility formula is called *arbitration-free* if its definition does not use the arbitration relation ar, i.e. the grammar in Equation (5.3) omits the ar relation. For example, in Figure 5.4, `RVC` and `CC` are arbitration-free while `PC` and `SC` are not.

Definition 5.6.2. *A consistency model is called* arbitration-free *w.r.t. an operation specification* OpSpec *if the visibility formulas contained in some normal form w.r.t.* OpSpec *are arbitration-free.*

Defining arbitration-free via a normal form removes "redundant" occurrences of the arbitration-order, i.e. visibility relations that employ ar but are vacuous w.r.t. OpSpec. We also show in Section 5.11 that for every basic consistency model CMod, if some normal form consists of arbitration-free visibility formulas, then this holds for any other normal form (this is actually proved for the more general class of consistency models defined in Section 5.8.1).

Theorem 5.6.3 (Basic Arbitration-Free Consistency ($\mathbf{AFC_0}$)). *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a basic storage specification. The following statements are equivalent:*

1. CMod *is arbitration-free w.r.t.* OpSpec,

2. *there exists an available* Spec*-implementation.*

In the following, we present a summary for the proof of AFC_0, which contains a series of lemmas. We refer the reader to Section 5.7 for a detailed proof. Lemmas 5.6.4 to 5.6.6 show that if CMod is arbitration-free then there exists an available Spec-implementation, whereas Lemma 5.6.7 is used to show the converse.

5.6.1 Arbitration-Freeness Implies Availability

Assume that CMod is arbitration-free w.r.t. OpSpec. We first show that CMod is weaker than $\mathtt{CC}$.

Lemma 5.6.4. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a basic storage specification. If* CMod *is arbitration-free w.r.t.* OpSpec*, then* CMod *is weaker than* $\mathtt{CC}$*.*

Proof Sketch. If CMod is arbitration-free, then every simple visibility formula v in a normal form of CMod does not use ar, i.e. it only uses so, wr and rb. By Definition 5.3.4, $\mathsf{so} \cup \mathsf{wr} \subseteq \mathsf{rb}$ in any abstract execution ξ. Hence, for every object x, $\mathsf{ctxt}_{\mathsf{CMod}}(r, [\xi, x]) \subseteq \mathsf{ctxt}_{\mathtt{CC}}(r, [\xi, x])$, i.e. $\mathsf{CMod} \preccurlyeq \mathtt{CC}$. □

Lemma 5.6.5 below implies that if a consistency model CMod is weaker than $\mathtt{CC}$, then any available $(\mathtt{CC}, \mathsf{OpSpec})$-implementation is also an available $(\mathsf{CMod}, \mathsf{OpSpec})$-implementation.

Lemma 5.6.5. *Let* OpSpec *be a basic operation specification, and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be a pair of basic consistency models s.t.* CMod_2 *is weaker than* CMod_1*. Any abstract execution valid w.r.t.* $(\mathsf{CMod}_2, \mathsf{OpSpec})$ *is also valid w.r.t.* $(\mathsf{CMod}_1, \mathsf{OpSpec})$*.*

Lemma 5.6.6 shows that there exists an available $(\mathtt{CC}, \mathsf{OpSpec})$-implementation, which concludes the proof of this direction.

Lemma 5.6.6. *Let* OpSpec *be a basic operation specification. There exists an available* $(\mathtt{CC}, \mathsf{OpSpec})$*-implementation.*

Proof Sketch. We define an available storage implementation of $(\mathtt{CC}, \mathsf{OpSpec})$ which is an abstraction of existing $\mathtt{CC}$ implementations [21, 80, 82, 22].

The storage implementation I_E describes a transition function associating events with the write-read relation obtained by computing the maximum writing event on its causal past (i.e. all write events that are already received in its replica). Each replica r maintains the causal past as follows: (1) every event invoked at r is added to r's causal past, (2) after every invocation, r broadcasts a message to all other replicas that contains its causal past, (3) whenever a replica r' receives this message, it adds the included causal past to its own. Sent messages are not required to be received before executing an invocation. The latter implies trivially that I_E is an available storage implementation. The validity w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ follows easily from the "transitive" communication of causal pasts between replicas. □

5.6.2 Availability Implies Arbitration-Freeness

We prove the contrapositive: if CMod is not arbitration-free, then no available Spec-implementation exists. Indeed, if CMod is not arbitration-free, every normal form CMod' of CMod contains a simple visibility formula involving ar (see Definition 5.6.2). By Lemma 5.6.7,

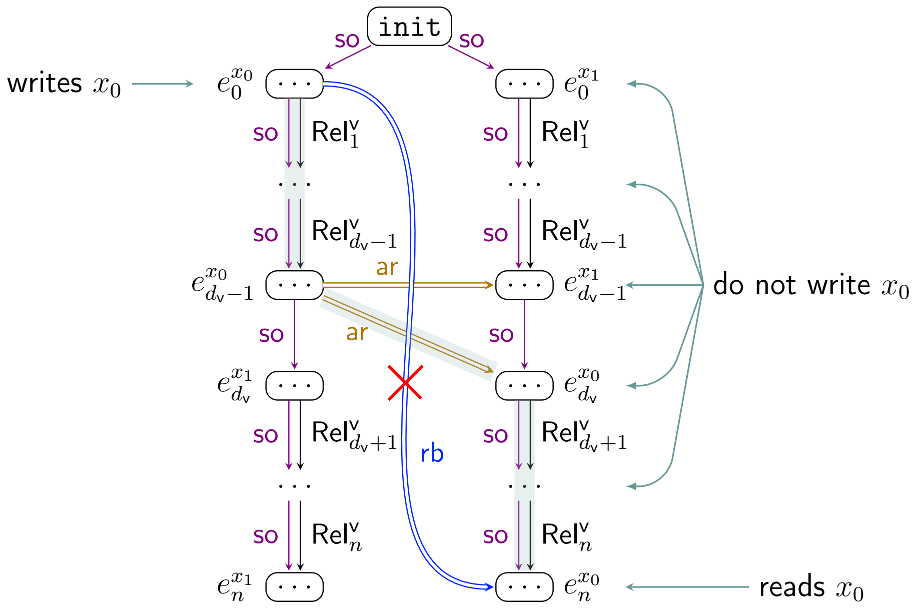

init
so
so
writes x_0
$e_0^{x_0}$
$e_0^{x_1}$
$\mathsf{Rel}_1^{\mathsf{v}}$
$\mathsf{Rel}_{d_{\mathsf{v}}-1}^{\mathsf{v}}$
ar
$e_{d_{\mathsf{v}}-1}^{x_0}$
$e_{d_{\mathsf{v}}-1}^{x_1}$
do not write x_0
$e_{d_{\mathsf{v}}}^{x_1}$
$e_{d_{\mathsf{v}}}^{x_0}$
$\mathsf{Rel}_{d_{\mathsf{v}}+1}^{\mathsf{v}}$
rb
$\mathsf{Rel}_n^{\mathsf{v}}$
$e_n^{x_1}$
$e_n^{x_0}$
reads x_0

Figure 5.5: Abstract execution of a trace without `receive` actions for the visibility formula v. If $i \neq d_{\mathsf{v}}$, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{Rel}_i^{\mathsf{v}}$ holds because the two events are executed at the same replica (recall that $\mathsf{so} \subseteq \mathsf{rb} \subseteq \mathsf{ar}$). If $(e_{d_{\mathsf{v}}-1}^{x_0}, e_{d_{\mathsf{v}}-1}^{x_1}) \in \mathsf{ar}$, then since $\mathsf{so} \subseteq \mathsf{ar}$ and ar is transitive, we get that $(e_{d_{\mathsf{v}}-1}^{x_0}, e_{d_{\mathsf{v}}}^{x_0}) \in \mathsf{Rel}_{d_{\mathsf{v}}}^{\mathsf{v}} = \mathsf{ar}$; and therefore, that $\mathsf{v}_{x_0}(e_0^{x_0}, e_n^{x_0})$ holds. However, in the absence of receives, $(e_0^{x_0}, e_n^{x_0}) \notin \mathsf{rb}$.

such a formula precludes the existence of an available $(\mathsf{CMod}', \mathsf{OpSpec})$-implementation. Consequently, there is no available $(\mathsf{CMod}, \mathsf{OpSpec})$-implementation, since any such implementation would also be an available $(\mathsf{CMod}', \mathsf{OpSpec})$-implementation – this is an easy observation as CMod is equivalent to CMod'.

Lemma 5.6.7. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a basic storage specification. Assume that* CMod *contains a simple visibility formula* v *which is non-vacuous w.r.t.* OpSpec*, such that for some* $i, 0 \leq i \leq \mathsf{len}(\mathsf{v})$*,* $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$*. Then, there is no available* $(\mathsf{CMod}, \mathsf{OpSpec})$*-implementation.*

Proof Sketch. We assume by contradiction that there is an available implementation I_E of Spec. . We use the visibility formula v to construct a specific program, which by the assumption, admits a trace (in the composition with this implementation) that contains no `receive` action. We show that any abstract execution induced by this trace, which is admissible by any available implementation of Spec, is not valid w.r.t. Spec.

The program P we construct generalizes the litmus programs presented in Figure 5.1. P uses two replicas r_0, r_1, two distinguished objects x_0, x_1 and a collection of events $e_i^{x_l}, 0 \leq i \leq n, l \in \{0, 1\}$. The events are used to “encode” two instances v_{x_0} and v_{x_1} of the visibility formula.

Let d_{v} be the largest index i s.t. $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$ (last occurrence of ar). Then, v is formed of two parts: the path of dependencies from ε_0 to $\varepsilon_{d_{\mathsf{v}}}$ which is not arbitration-free, and the suffix from $\varepsilon_{d_\mathsf{v}}$ up to $\varepsilon_{\mathsf{len}(\mathsf{v})}$, the arbitration-free part. Thus, v is of the form:

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_n) ::= \exists \varepsilon_1, \ldots, \varepsilon_{n-1}. \bigwedge_{i=1}^{n} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^\mathsf{v} \ \wedge \varepsilon_0 \text{ writes } x \wedge \mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset$$

where $n = \mathsf{len}(\mathsf{v})$, $\mathsf{Rel}_i^\mathsf{v} \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}, \mathsf{ar}\}$ for $i < d_v$, $\mathsf{Rel}_{d_\mathsf{v}}^\mathsf{v} = \mathsf{ar}$, and $\mathsf{Rel}_i^\mathsf{v} \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}\}$ for $i > d_\mathsf{v}$.

Replica r_l executes first events $e_i^{x_l}$ with $i < d_\mathsf{v}$ and then, events $e_i^{x_{1-l}}$ with $i \geq d_\mathsf{v}$ – the replica r_l executes the non arbitration-free part of v for object x_l and the arbitration-free suffix of v for x_{1-l}. All events in replica r_l access (read and/or write) object x_l except for $e_n^{x_l}$ which reads x_{1-l}. For ensuring that $\mathsf{v}_x(e_0^{x_l}, \ldots e_n^{x_l})$ holds in an induced abstract execution of a trace without `receive` actions, we require that if $\mathsf{Rel}_i^\mathsf{v} = \mathsf{wr}$, then $e_{i-1}^{x_l}$ is a write event and $e_i^{x_l}$ is a read event. Figure 5.5 exhibits a diagram of such execution.

Example 5.6.8. *We illustrate the construction for Prefix Consistency (*`PC`*) and a Key-Value store with* `PUT` *and* `GET` *operations (their specification is defined in Section 5.4.2).* `PC` *can be defined as the following set of* simple *visibility formulas (obtained from* Prefix *in Figure 5.4c):*

$$\begin{aligned}
\mathsf{v}_x^1(\varepsilon_0, \varepsilon_1) &::= \varepsilon_0 \text{ writes } x \ \wedge \ \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \ \wedge \ (\varepsilon_0, \varepsilon_1) \in \mathsf{so} \\
\mathsf{v}_x^2(\varepsilon_0, \varepsilon_1) &::= \varepsilon_0 \text{ writes } x \ \wedge \ \mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \ \wedge \ (\varepsilon_0, \varepsilon_1) \in \mathsf{wr} \\
\mathsf{v}_x^3(\varepsilon_0, \varepsilon_2) &::= \exists \varepsilon_1. \ \varepsilon_0 \text{ writes } x \ \wedge \ \mathsf{wr}_x^{-1}(\varepsilon_2) \neq \emptyset \ \wedge \ (\varepsilon_0, \varepsilon_1) \in \mathsf{ar} \ \wedge \ (\varepsilon_1, \varepsilon_2) \in \mathsf{so} \\
\mathsf{v}_x^4(\varepsilon_0, \varepsilon_2) &::= \exists \varepsilon_1. \ \varepsilon_0 \text{ writes } x \ \wedge \ \mathsf{wr}_x^{-1}(\varepsilon_2) \neq \emptyset \ \wedge \ (\varepsilon_0, \varepsilon_1) \in \mathsf{ar} \ \wedge \ (\varepsilon_1, \varepsilon_2) \in \mathsf{wr}
\end{aligned} \tag{5.6}$$

Observe that v_x^4 is vacuous w.r.t. the specification of `PUT` *and* `GET` *since it implies that ε_2 reads from multiple events, and* `PUT` *and* `GET` *read a single object at a time. Thus, the normal form of* `PC` *w.r.t. the specification of* `PUT` *and* `GET` *contains only the first three visibility formulas above.*

The only visibility formula which is not arbitration-free is v_x^3. We have that the index $d_\mathsf{v} = 1$ and we consider the following types of events:

$$e_0^{x_0} : \texttt{PUT}(x_0, \_), e_1^{x_0} : \texttt{PUT}(x_1, \_), e_2^{x_0} : \texttt{GET}(x_0)$$
$$e_0^{x_1} : \texttt{PUT}(x_1, \_), e_1^{x_1} : \texttt{PUT}(x_0, \_), e_2^{x_1} : \texttt{GET}(x_1)$$

Replica r_0 executes $e_0^{x_0}$ and then $e_1^{x_1}$ and $e_2^{x_1}$. Replica r_1 executes $e_0^{x_1}$ and then $e_1^{x_0}$ and $e_2^{x_0}$.

Given such a program P, the proof proceeds as follows:

1. There exists a finite trace t of $P \parallel I_E$ that contains no receive action (Lemma 5.7.5): Since I_E is available, it can always delay receiving messages, and execute other actions instead. Then, as P is a finite program, such an execution must be finite.

2. The trace t induces a history $h_\mathsf{v} = (E, \mathsf{so}, \mathsf{wr})$ and an abstract execution $\xi_\mathsf{v} = (h, \mathsf{rb}, \mathsf{ar})$ where $\mathsf{rb} = \mathsf{so}$ (ar is arbitrary as long as $\mathsf{rb} \subseteq \mathsf{ar}$). As I_E is valid w.r.t. Spec, ξ_v is valid w.r.t. Spec. Next, we prove that since $\mathsf{rb} = \mathsf{so}$, events in ξ_v read the latest value w.r.t. so written on their associated object in ξ_v (Lemma 5.7.6). In particular, we deduce that all traces of P without `receive` events induce the same history and therefore, the induced history does not change when the induced arbitration order changes.

3. Since ar is a total order, either $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$ or $(e^{x_1}_{d_\mathsf{v}-1}, e^{x_0}_{d_\mathsf{v}-1}) \in \mathsf{ar}$. W.l.o.g., assume that $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$. By Lemma 5.7.7, we deduce that $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_n, [\xi_\mathsf{v}, \mathsf{CMod}])$. The proof is explained in Figure 5.5: if $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$, then all events $e^{x_0}_i$ form a path in such way that $\mathsf{v}_{x_0}(e^{x_0}_0, \ldots e^{x_0}_n)$ holds in ξ_v.

4. Since $e^{x_0}_n$ is the only event at r_1 that reads or writes x_0 and events in ξ_v read the latests values w.r.t. so in ξ_v, we deduce that $e^{x_0}_n$ reads x_0 from `init`. However, as $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_n, [\xi_\mathsf{v}, \mathsf{CMod}])$ and `init` precedes $e^{x_0}_0$ in arbitration order, we deduce that $e^{x_0}_n$ does not read the latest value w.r.t. ar, i.e. $\mathsf{rspec}(e^{x_0}_n) \downarrow$ but $\mathsf{wr}^{-1}_{x_0}(e^{x_0}_n) \neq \{\max_{\mathsf{ar}} \mathsf{ctxt}_{x_0}(e^{x_0}_n, [\xi_\mathsf{v}, \mathsf{CMod}])\}$. Therefore, ξ_v is not valid w.r.t. Spec (see Definition 5.4.2). This contradicts the hypothesis that I_E is an implementation of Spec. $\square$

The corollary below is a direct consequence of Theorem 5.6.3 and Lemma 5.6.4.

Corollary 5.6.9. *Let* OpSpec *be a basic operation specification. The strongest consistency model* CMod *for which* $(\mathsf{CMod}, \mathsf{OpSpec})$ *admits an available implementation is* $\mathtt{CC}$.

5.7 Proof of the Basic Arbitration-Free Consistency Theorem

The proof ot the Basic Arbitration-Free Consistency Theorem relies on several results from Section 5.11 about consistency models in normal form. Nevertheless, basic consistency models are a subclass of the consistency models presented in Section 5.8.

Let in the folloeing $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ be a basic storage specification. We show that there exists an available Spec-implementation iff CMod is arbitration-free w.r.t. OpSpec.

5.7.1 Arbitration-Freeness Implies Availability

As discussed in Section 5.6, the proof of such result is decomposed in three steps:

1. We show that arbitration-free consistency models w.r.t. OpSpec are weaker than $\mathtt{CC}$ (Lemma 5.6.4).

2. We deduce that available $(\mathtt{CC}, \mathsf{OpSpec})$-implementations are also available $(\mathsf{CMod}, \mathsf{OpSpec})$-implementations as an immediate consequence of Lemma 5.6.5.

3. We prove that there exists available $(\mathtt{CC}, \mathsf{OpSpec})$-implementations (Lemma 5.6.6).

Lemma 5.6.4. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a basic storage specification. If* CMod *is arbitration-free w.r.t.* OpSpec*, then* CMod *is weaker than* $\mathtt{CC}$.

Proof. For showing that CMod is weaker than $\mathtt{CC}$, let $h = (E, \mathsf{so}, \mathsf{wr})$ be a history and $\xi = (h, \mathsf{rb}, \mathsf{ar})$ be an abstract execution of h valid w.r.t. Spec. Let n be a consistency model in normal form that is OpSpec-equivalent to CMod. By Theorem 5.11.1, such model always exists. As CMod is arbitration-free, every visibility formula $v \in n$ is arbitration-free. We conclude the result by showing that $n \preccurlyeq \mathtt{CC}$, i.e. showing that for every object x and every pair of distinct events $e, e' \in E$, if $v_x(e, e')$ holds in ξ then $v^{\mathtt{CC}}_x(e, e')$ holds in ξ as well; where $v^{\mathtt{CC}}$ is Causal, the visibility formula of $\mathtt{CC}$ (Figure 5.4b).

First, as $v_x(e, e')$ holds in ξ, e writes x in ξ and $\mathsf{wr}_x^{-1}(e) \neq \emptyset$. Moreover, as v is simple, for every $i, 1 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{\mathsf{v}} \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}\}$. By Property 2 of Definition 5.3.4, we deduce that $(e, e') \in \mathsf{rb}^+$. Altogether, we conclude that $v_x^{\mathsf{CC}}(e, e')$ holds in ξ. □

Lemma 5.6.6. *Let* OpSpec *be a basic operation specification. There exists an available* $(\mathsf{CC}, \mathsf{OpSpec})$*-implementation.*

Proof. We define an available implementation of $\mathsf{Spec}^{\mathsf{CC}} = (\mathsf{CC}, \mathsf{OpSpec})$.

As discussed in Section 5.5, any implementation $I_E = (S_{\mathsf{i}}, A_{\mathsf{i}}, s_0^{\mathsf{i}}, \Delta_{\mathsf{i}})$ can be characterized by describing its set of states S_{i}, its actions A_{i}, its initial state σ_0^{i} and its transition function Δ_{i}.

First, we define S_{i} as the set of possible values that each object may have; and the declare the initial state any possible state in S_{i}. Next, we define A_{i} via the synchronized actions $\mathsf{Events} \times (\mathsf{Keys} \times \mathsf{Events} \cup \{\emptyset\})$, as well as the local actions $\mathtt{send}$ and $\mathtt{receive}$. We assume local actions are defined in a similar way to Events, as tuples $a = (\mathsf{id}, \mathsf{r}, \mathsf{op}, \mathsf{wval}, \mathsf{m})$, where id is an action identifier, r is a replica identifier, op an operation identifier, wval is a (partial) mapping associating each object x with a value v that this event writes to x, and m is additional metadata of the action. As for events, we use $\mathtt{id}(a)$, $\mathtt{rep}(a)$, $\mathtt{op}(a)$, $\mathtt{wval}(a)$ and $\mathtt{md}(a)$ for indicating the identifier, replica, operation, write-value mapping and metadata of an action a respectively.

For describing its transition function, we rely on the definition of CC. As we design $(S_{\mathsf{i}}, A_{\mathsf{i}}, s_0^{\mathsf{i}}, \Delta_{\mathsf{i}})$ to be an available $\mathsf{Spec}^{\mathsf{CC}}$-implementation, we require that any induced abstract execution must be valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$. However, Definition 5.4.2 describes validity “a posteriori”, i.e. validity can only be checked once the event is executed; while transition functions describe validity “a priori”, i.e. describe a procedure to compute a write-read of a given, not yet added event. For solving this issue, we observe that under CC, that the context of an event e belonging to a synchronized action $a = (e, m)$ only depends on (a) the transitive set of received actions before the last action in its replica and (b) the synchronized actions executed in its own replica. Ensuring transitive communication, i.e. ensuring that every send action on replica r transmits information about all synchronized actions executed or received on replica r before such $\mathtt{send}$ action suffices to provide CC.

More in detail, for describing the transition function $\Delta_{\mathsf{i}}(t, a)$, we require that (1) a is not present in t and (2) transitive communication is ensured. Also, we require a third condition depending on the type of a:

- if a is a synchronized action, we require that (3a) if a represents a read operation, $a = (e, m)$, then e must read from the latest writing event w.r.t. ar (which coincides with the trace order) received before l_r^t,
- if a is a $\mathtt{send}$ action, then (3b) it precedes a synchronized action, and
- if a is a $\mathtt{receive}$ action, then (3c) there exists a unique preceding $\mathtt{send}$ action that matches it.

where $r = \mathtt{rep}(a)$ and l_r^t to the last action in trace t whose replica is r.

On one hand, (1) ensures that $\Delta_{\mathsf{i}}(t,e)$ is well-defined, i.e. in every trace of Δ_{i}, each action contains each action exactly once. On the other hand, (2) and (3a) ensure that I_E is a $\mathsf{Spec}^{\mathsf{CC}}$-storage implementation while (3b) and (3c) ensure that I_E is an available storage implementation.

Formally, $\Delta_{\mathsf{i}}(t,a)\downarrow$ if and only if $a \notin t$ and $\mathsf{sat}(t,a)$ holds; and in such case $\Delta_{\mathsf{i}}(t,a) = t \oplus a$. The predicate $\mathsf{sat}(t,a)$ is described in Equation (5.7).

$$\mathsf{sat}(t,a) = \begin{cases} a = (e, M_t(e)) & \text{if } \mathsf{op}(a) \neq \mathtt{send},\mathtt{receive} \\ \mathsf{sendIfData}(t,a), & \text{if } \mathsf{op}(a) = \mathtt{send} \\ \ \mathsf{sendAllData}(t,a), & \\ \ \text{and } \mathsf{maxSend}(t,a) & \\ \mathsf{minRcv}(t,a), & \text{if } \mathsf{op}(a) = \mathtt{receive} \\ \ \text{and } \mathsf{maxRcv}(t,a) & \end{cases} \tag{5.7}$$

where $M_t(e)$ is the mapping assigning to the objext $x = \mathsf{obj}(e)$ the last event that writes on x received by e, formally defined using Equations (5.8) and (5.9); and the predicates $\mathsf{sendIfData}$, $\mathsf{sendAllData}$, $\mathsf{maxSend}$, minRcv and maxRcv are defined in Equations (5.10) to (5.13).

$$\begin{aligned} M_t(e) &= \left[x \mapsto \begin{cases} \{\max_{\mathsf{ar}^t_e} E^x_t(e)\} & \text{if } x = \mathsf{obj}(e) \\ \emptyset & \text{otherwise} \end{cases} \right]_{x \in \mathsf{Keys}} \\ E^x_t(e) &= \left\{ e' \;\middle|\; \begin{array}{l} e' \in \mathsf{Events} \cap t \ \wedge \ e' \text{ writes } x \text{ in } \mathtt{exec}(t) \ \wedge \\ (\mathtt{rep}(e') = \mathtt{rep}(e) \vee \mathsf{rec}_t(e',e)) \end{array} \right\} \\ \mathsf{ar}^t_e &= \mathsf{ar}_{\restriction E^x_t(e) \times E^x_t(e)} \end{aligned} \tag{5.8}$$

$$\mathsf{rec}_t(e',e) = \exists r,s \in t \text{ s.t. } \bigwedge \begin{array}{l} \mathsf{op}(r) = \mathtt{receive}, \mathtt{rep}(r) = \mathtt{rep}(e), \\ \mathsf{op}(s) = \mathtt{send}, \mathtt{rep}(s) = \mathtt{rep}(e'), \\ \mathsf{rb}\text{-}\mathtt{Set}(s) = \mathsf{rb}\text{-}\mathtt{Set}(r), e' <_t s <_t r < e' \end{array} \tag{5.9}$$

$$\mathsf{sendIfData}(t,a) ::= \mathsf{op}(a'') \neq \mathtt{send} \tag{5.10}$$

$$\text{where } a'' = \max_{<_t} \left\{ a' \in t \;\middle|\; \mathtt{rep}(a') = \mathtt{rep}(a) \ \wedge \ \mathsf{op}(a') \neq \mathtt{receive} \right\}$$

$$\mathsf{sendAllData}(t,a) ::= \forall a' \in t.\mathtt{rep}(a') = \mathtt{rep}(a) \wedge \mathsf{op}(a') \neq \mathtt{send} \implies \mathsf{RV}^x_{a'} \subseteq \mathsf{rb}\text{-}\mathtt{Set}(a) \tag{5.11}$$

$$\text{where } \mathsf{RV}^x_{a'} = \begin{cases} \{e\} & \text{if } \mathsf{op}(a') \neq \mathtt{send},\mathtt{receive} \ \wedge \ a' = (e, \_) \\ \mathsf{rb}\text{-}\mathtt{Set}(a') & \text{if } \mathsf{op}(a') = \mathtt{receive} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{maxSend}(t,a) ::= \nexists a' \in t.\mathsf{op}(a') = \mathtt{send} \wedge \mathsf{rb}\text{-}\mathtt{Set}(a) = \mathsf{rb}\text{-}\mathtt{Set}(a') \tag{5.12}$$

$$\mathsf{minRcv}(t,a) ::= \exists a' \in t.\mathsf{op}(a') = \mathtt{send} \wedge \mathsf{rb}\text{-}\mathtt{Set}(a) = \mathsf{rb}\text{-}\mathtt{Set}(a') \tag{5.13}$$

$$\mathsf{maxRcv}(t,a) ::= \nexists a' \in t.\, \mathsf{op}(a') = \mathtt{receive} \ \wedge \ \mathtt{rep}(a) = \mathtt{rep}(a') \ \wedge \ \mathsf{rb}\text{-}\mathtt{Set}(a) = \mathsf{rb}\text{-}\mathtt{Set}(a') \tag{5.14}$$

Note that as I_E contains `send` and `receive` actions, as well as events along with their write-read dependencies, I_E is a storage implementation. For proving that I_E is the searched implementation, we introduce the following notation: for a trace t and an event $e \in t$, $\mathsf{prefix}(t, e)$ to the trace s.t. $\Delta(\mathsf{prefix}(t, e), e)$ is a prefix of t.

The rest of the proof, showing that I_E is an available $\mathsf{Spec}^{\mathsf{CC}}$-implementation, is a consequence of Lemmas 5.7.1 to 5.7.3. $\square$

Lemma 5.7.1. *The implementation I_E is an $\mathsf{Spec}^{\mathsf{CC}}$-implementation.*

Proof. Let $P_E = (S_\mathsf{p}, A_\mathsf{p}, s_0^\mathsf{p}, \Delta_\mathsf{p})$ be a program. We prove by induction on the length of all traces in $\mathcal{T}_{P_E \| I_E}$ that any trace t is feasible and its induced abstract execution is valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$. The base case, when $t = \{(\mathtt{init}_{P_E}, \mathtt{init}_{I_E})\}$ is immediate as t contains exactly one event that does not read any object. Hence, let us assume that for any trace $t' \in \mathcal{T}_{P_E \| I_E}$ of at most length k, $\mathtt{exec}(t')$ is valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$; and let us show that for any trace t of length $k+1$, $\mathtt{exec}(t)$ is also valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$. Let $h = (E, \mathsf{so}, \mathsf{wr})$ and $\xi = (h, \mathsf{rb}, \mathsf{ar})$ be respectively the induced history $\mathsf{history}(t)$ and the induced abstract execution $\mathtt{exec}(t)$ where ar coincides with the trace order. We denote sr to the induced order between send-receive actions with the same rb-Set on t. Before proving that ξ is valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$, we show that t is feasible, i.e. ξ satisfies Definition 5.3.4.

- $\underline{\mathsf{rb} = \mathsf{rb}; \mathsf{so}^*}$: This is immediate by the definition of induced receive-before.
- $\underline{\mathsf{wr} \cup \mathsf{so} \subseteq \mathsf{rb}}$: By definition of rb, $\mathsf{so} \subseteq \mathsf{rb}$, so we focus on proving that $\mathsf{wr} \subseteq \mathsf{rb}$. Let w, r be events and x be an object s.t. $(w, r) \in \mathsf{wr}_x$. In such case, there is a pair of actions a_r, a_w s.t. $r \in a_r$, $w \in a_w$ and $w \in \mathsf{wr}\text{-}\mathtt{Set}(a_r)(x)$. Hence, $\{w\} = \max_{\mathsf{ar}_e^t} E_t^x(e)$. We deduce then that $\mathsf{rec}_t(w, r)$ must hold; which implies that there exists a `send` action s and a `receive` action v s.t. $\mathsf{rb}\text{-}\mathtt{Set}(s) = \mathsf{rb}\text{-}\mathtt{Set}(v)$ and $w <_t s <_t v <_t r$. By $\mathsf{sendAllData}$ predicate, $w \in \mathsf{rb}\text{-}\mathtt{Set}(s)$. As $\mathsf{rb}\text{-}\mathtt{Set}(s) = \mathsf{rb}\text{-}\mathtt{Set}(v)$, $w \in \mathsf{rb}\text{-}\mathtt{Set}(v)$. By the definition of induced abstract execution, $(w, r) \in \mathsf{rb}$.
- $\underline{\mathsf{rb} \subseteq \mathsf{ar}}$: For proving that $\mathsf{rb} \subseteq \mathsf{ar}$, as rb can be derived by sr and so, it suffices to prove that both $\mathsf{so}, \mathsf{sr} \subseteq \mathsf{ar}$. First, as so is the partial order induced by the total order $<_t$ on actions executed on the same replica, $\mathsf{so} \subseteq \mathsf{ar}$.

  Next, for proving that $\mathsf{sr} \subseteq \mathsf{ar}$, let s be a `send` action and let v be a `receive` action s.t. $(s, v) \in \mathsf{sr}$. Let us consider $p_v^t = \mathsf{prefix}(t, v)$ be the prefix of t before v. On one hand, as p_v^t is a prefix of t', $\Delta_\mathsf{i}(p_v^t, v) \downarrow$. In particular, $\mathsf{minRcv}(p_v^t, v)$ holds; so there is a `send` action s' in p_v^t s.t. $\mathsf{rb}\text{-}\mathtt{Set}(s') = \mathsf{rb}\text{-}\mathtt{Set}(v)$. We show that $s' = s$. Otherwise, then w.l.o.g. $s <_t s'$. Note that $\Delta_\mathsf{i}(\mathsf{prefix}(t, s'), s') \downarrow$ as $\mathsf{prefix}(t, s') \oplus s'$ is a prefix of t'. In such case, $\mathsf{maxSend}(\mathsf{prefix}(t, s'), s')$ does not hold; which is impossible as $\Delta_\mathsf{i}(\mathsf{prefix}(t, s'), s') \downarrow$. Therefore, $s = s'$. As $s' \in p_v^t$, s precedes v in t; so $(s, v) \in \mathsf{ar}$.

After proving that t is feasible, we show that ξ is valid w.r.t. $\mathsf{Spec}^{\mathsf{CC}}$. By Definition 5.4.2, we need to show that for every event r and object x, if $\mathsf{rspec}(r) \uparrow$, $\mathsf{wr}_x^{-1}(r) = \emptyset$, and otherwise, $\mathsf{wr}_x^{-1}(r) = \{\max_{\mathsf{ar}} \mathsf{ctxt}_x(r, [\xi, \mathsf{CC}])\}$. Let r be a read event, x be the object it affects and $p = \mathsf{prefix}(t, r)$. We know by Equation (5.8) that $\mathsf{wr}_x^{-1}(r) = \{\max_{\mathsf{ar}_r^p} E_p^x(r)\}$. Observe then that by Equation (5.8) and rb's definition, $E_p^x(r) = \mathsf{ctxt}_x(r, [t, \mathtt{CC}])$. Thus, we conclude that $\mathsf{wr}_x^{-1}(r) = \{\max_{\mathsf{ar}} \mathsf{ctxt}_x(r, [\xi, \mathtt{CC}])\}$. □

Lemma 5.7.2. *For every program P_E and every trace t of $I_E \parallel P_E$, there is no replica waiting in t.*

Proof. Let $P_E = (S_{\mathsf{p}}, A_{\mathsf{p}}, s_0^{\mathsf{p}}, \Delta_{\mathsf{p}})$ be a program, $r \in \mathsf{Reps}$ be a replica and $t \in \mathcal{T}_{P_E \parallel I_E}$ be a reachable trace. Let also be $t_1 \in \mathcal{T}_{P_E}$ and $t_2 \in \mathcal{T}_{I_E}$ traces s.t. $t = (t_1, t_2)$. To prove that r is not waiting in t, let us suppose that there exists an event $e \in \mathsf{Events}_{P_E}$ s.t. $\mathtt{op}(e) \neq \mathtt{end}$, $\mathtt{rep}(e) = r$ and $\Delta_{P_E}(t_1, e) \downarrow$, and let us prove that there exists a non-$\mathtt{receive}$ action a s.t. $\Delta_{I_E \parallel P_E}(t, a) \downarrow$.

Let a be the action $(e, M_t(e))$; where $M_t(e)$ is described using Equation (5.8). We observe that as $\Delta_{P_E}(t_1, e) \downarrow$, $\Delta_{P_E \parallel I_E}(t, \mathsf{ex}) \downarrow$. Moreover, $\mathtt{op}(a) \neq \mathtt{receive}$. Hence, r is not waiting in t; so I_E is available. □

Lemma 5.7.3. *For every finite program P_E, the composition $I_E \parallel P_E$ is also finite.*

Proof. Let $P_E = (S_{\mathsf{p}}, A_{\mathsf{p}}, s_0^{\mathsf{p}}, \Delta_{\mathsf{p}})$ be a finite program. The implementation I_E is conditionally finite on P_E if for every trace $t \in \mathcal{T}_{P_E \parallel I_E}$ there exists a constant $k_t \in \mathtt{N}$ s.t. $\mathsf{len}(t) \leq k_t$. Let thus $t \in \mathcal{T}_{P_E \parallel I_E}, t_1 \in \mathcal{T}_{P_E}, t_2 \in \mathcal{T}_{I_E}$ be traces s.t. $t = (t_1, t_2)$. As P_E is finite, the length of t_1, $\mathsf{len}(t_1)$, is finite. We show that $k_t ::= 3 \cdot \mathsf{len}(t_1)$ is the constant we search.

Three cases arise, depending on the type of action we consider. First, by maxRcv predicate, the number of $\mathtt{receive}$ actions coincides with the number of $\mathtt{receive}$ actions with distinct metadata; which by minRcv, is bounded by the number of $\mathtt{send}$ actions in the trace. Then, by $\mathsf{sendIfData}$, the number of $\mathtt{send}$ actions is bounded by the number of synchronized actions. Finally, by the parallel composition definition, the number of synchronized actions in t and t_1 coincide; so such number is bounded by $\mathsf{len}(t_1)$. Altogether, we deduce that $\mathsf{len}(t) \leq 3 \cdot \mathsf{len}(t_1) = k_t$. □

5.7.2 Availability Implies Arbitration-Freeness

As explained in Section 5.6, we prove the contrapositive: if CMod is not arbitration-free, then no available Spec-implementation exists. Indeed, if CMod is not arbitration-free, every normal form CMod' of CMod contains a simple visibility formula involving ar (see Definition 5.6.2). By Lemma 5.6.7, such a formula precludes the existence of an available $(\mathsf{CMod}', \mathsf{OpSpec})$-implementation. Consequently, there is no available $(\mathsf{CMod}, \mathsf{OpSpec})$-implementation, since any such implementation would also be an available $(\mathsf{CMod}', \mathsf{OpSpec})$-implementation – this is an easy observation as CMod is equivalent to CMod' (see Theorem 5.11.1).

Proof. We assume by contradiction that there is an available implementation I_E of Spec but CMod contains a visibility formula v s.t. for some $i, 0 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$. We use the latter fact to construct a specific program, which by the assumption, admits a trace (in the composition with this implementation) that contains no $\mathtt{receive}$ action. We show that any abstract execution induced by this trace, which is admissible by any available implementation of Spec, is not valid w.r.t. Spec. This contradicts the hypothesis.

The program P we construct generalizes the litmus program presented in Figure 5.1. P uses two replicas r_0, r_1, two distinguished objects x_0, x_1 and a collection of events $e_i^{x_l}, 0 \leq i \leq \mathsf{len}(v), l \in \{0, 1\}$. The events are used to "encode" two instances v_{x_0} and v_{x_1} of the visibility formula.

Let d_v be the largest index i s.t. $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$ (last occurrence of ar). Then, v is formed of two parts: the path of dependencies from ε_0 to ε_{d_v} which is not arbitration-free, and the suffix from ε_{d_v} up to $\varepsilon_{\mathsf{len}(v)}$, the arbitration-free part. Thus, v is of the form:

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_{\mathsf{len}(v)}) ::= \exists \varepsilon_1, \ldots, \varepsilon_{n-1}. \bigwedge_{i=1}^{\mathsf{len}(v)} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^{\mathsf{v}} \ \wedge \varepsilon_0 \ \mathsf{writes}\ x \wedge \mathsf{wr}_x^{-1}(\varepsilon_{\mathsf{len}(v)}) \neq \emptyset$$

where $\mathsf{Rel}_i^v \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}, \mathsf{ar}\}$, for all $i < d_v$, $\mathsf{Rel}_{d_v}^v = \mathsf{ar}$, and $\mathsf{Rel}_i^v \in \{\mathsf{so}, \mathsf{wr}, \mathsf{rb}\}$ for all $i > d_v$.

In the construction, we require that replica r_l executes events $e_i^{x_l}$ if $i < d_v$ and events $e_i^{x_{1-l}}$ otherwise – the replica r_l executes the non arbitration-free part of v for object x_l and the arbitration-free suffix of v for x_{1-l}. All objects in replica r_l access (read and/or write) x_l except $e_{\mathsf{len}(v)}^{x_l}$, which access with x_{1-l}. We denote by $\tilde{x}_i^{x_l}$ to the unique object that event $e_i^{x_l}$ reads and/or writes.

More in detail, we construct a set of events, E^i, histories, $h^i = (E^i, \mathsf{so}^i, \mathsf{wr}^i)$, and executions, $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$, $0 \leq i \leq \mathsf{len}(v)$ inductively, starting from an initial event `init`, and incorporating at each time a pair of new events, $e_i^{x_0}$ and $e_i^{x_1}$. For simplifying notation, we use the convention $\texttt{init} = e_{-1}^{x_0} = e_{-1}^{x_1}$.

For the inductive step, we assume that the abstract execution $\xi^{i-1} = (h^{i-1}, \mathsf{rb}^{i-1}, \mathsf{ar}^{-1})$ associated to the history $h^{i-1} = (E^{i-1}, \mathsf{so}^{i-1}, \mathsf{wr}^{i-1})$ contains events $e_{-1}^{x_0} \ldots e_{i-1}^{x_0}, e_{i-1}^{x_1}$ and is well-defined (satisfies Definition 5.3.4) and we construct the history h^i and the abstract execution ξ^i.

We distinguish between cases depending on the value i:

- $\underline{i = 0}$: In this case, we consider e_0 be an event s.t. $\mathsf{wspec}(e_0^{x_l})(\mathtt{wval}(\mathtt{init})(\tilde{x}_i^{x_l}))\downarrow$.
- $\underline{0 < i < \mathsf{len}(v),\ \mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr} \text{ and } \mathsf{Rel}_{i+1}^{\mathsf{v}} = \mathsf{wr}}$: In this case, it is easy to see that by Proposition 5.11.11, OpSpec allows atomic read-write events. We consider $e_i^{x_l}$ be an event s.t. $\mathsf{rspec}(e_i^{x_l})\downarrow$ and $\mathsf{wspec}(w_i^{x_l})(\mathtt{value}_{\mathsf{wr}^{i-1}}(w_i^{x_l}, \tilde{x}_i^{x_l}))\downarrow$.
- $\underline{0 < i < \mathsf{len}(v) \text{ and } \mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr} \text{ and } \mathsf{Rel}_{i+1}^{\mathsf{v}} = \mathsf{wr}}$: In this case, if OpSpec allows unconditional writes, then we select $e_i^{x_l}$ as an unconditional write event on object $\tilde{x}_i^{x_l}$. Otherwise, we select event $e_i^{x_l}$ s.t. $\mathsf{rspec}(e_i^{x_l})\downarrow$ and $\mathsf{wspec}(e_i^{x_l})(\mathtt{wval}(w_i^{x_l})(\tilde{x}_i^{x_l}))\downarrow$.
- $\underline{0 < i < \mathsf{len}(v) \text{ and } \mathsf{Rel}_{i+1}^{\mathsf{v}} \neq \mathsf{wr}}$: In this case, we select $e_i^{x_l}$ to not write $\tilde{x}_i^{x_l}$ unless it is necessary. If OpSpec allows read events that are not write events, or if allows conditional atomic read-write events, we select $e_i^{x_l}$ as an event such that $\mathsf{rspec}(e_i^{x_l})\downarrow$ but $\mathsf{wspec}(e_i^{x_l})(\mathtt{wval}(w_i^{x_l})(\tilde{x}_i^{x_l}))\uparrow$. Otherwise, we select event e_i such that $\mathsf{rspec}(e_i^{x_l})\downarrow$ and $\mathsf{wspec}(e_i^{x_l})(\mathtt{wval}(w_i^{x_l})(\tilde{x}_i^{x_l}))\downarrow$.
- $\underline{i = \mathsf{len}(v)}$: In this case, we consider $e_{\mathsf{len}(v)}^{x_l}$ to be an event that reads object $\tilde{x}_i^{x_l}$, i.e. $\mathsf{rspec}(e_{\mathsf{len}(v)}^{x_l})\downarrow$.

where $l \in \{0,1\}$ and $w_i^{x_l} = \max_{\mathsf{ar}^{i-1}}\{e \in E^{i-1} \mid \mathsf{wspec}(e)(\mathsf{obj}(e_i^{x_l})) \downarrow \wedge (e, e_i^{x_l}) \in \mathsf{so}^i\}$. We note that as `init` writes on every object, $w_i^{x_l}$ is well-defined.

First of all, observe that event $e_i^{x_l}$ is well-defined thanks to Lemma 5.7.4 and the assumptions on OpSpec (Section 5.4.3). We denote $E^i = E^{i-1} \cup \{e_i^{x_0}, e_i^{x_1}\}$. We observe that w.l.o.g., we can assume that the $\mathtt{id}(e_i^{x_0})$ is bigger than every identifier of an event in E^{i-1} and that $\mathtt{id}(e_i^{x_0}) < \mathtt{id}(e_i^{x_1})$.

We conclude the description of h^i and ξ^i by specifying the relations $\mathsf{so}^i, \mathsf{wr}^i, \mathsf{rb}^i, \mathsf{ar}^i$. We require that so^i (resp. wr^i, rb^i, ar^i) contains so^{i-1} (resp. wr^{i-1}, rb^{i-1}, ar^{i-1}). Also, we require additional constrains on them due to event e_i:

- $\underline{\mathsf{so}^i}$: We require that $(e, e_i^{x_l}) \in \mathsf{so}^i$ iff $\mathtt{rep}(e) = \mathtt{rep}(e_i^{x_l})$; as well as $(\mathtt{init}, e_i^{x_l}) \in \mathsf{so}^i$.
- $\underline{\mathsf{wr}^i}$: If $e_i^{x_l}$ is not a read event, we require that ${\mathsf{wr}^i_{x_i}}^{-1}(e_i^{x_l}) \neq \emptyset$. Otherwise, we require that $(\{w_i^{x_l}\}, e_i^{x_l}) \in \mathsf{wr}^i_{x_i}$.
- $\underline{\mathsf{rb}^i}$: We require that $\mathsf{rb}^i = \mathsf{so}^i$.
- $\underline{\mathsf{ar}^i}$: We impose that for every event $e \in E^i$, $(e, e_i^{x_l}) \in \mathsf{ar}^i$. Also, we impose that $(e_i^{x_0}, e_i^{x_1}) \in \mathsf{ar}^i$.

Then, we define $\mathsf{Events_p} = E^{\mathsf{len}(v)}$ as the set our program employs. The set $\mathsf{Events_p}$ induces the set of traces $\mathcal{T}_\mathsf{p}$.

We define the program $P = (S_\mathsf{p}, A_\mathsf{p}, s_0^\mathsf{p}, \Delta_\mathsf{p})$, where $\mathtt{init}_\mathsf{p} = \mathtt{init}$ and Δ_p is the transition function defined as follows: for every trace $t \in \mathcal{T}_\mathsf{p}$ and event $e \in \mathsf{Events_p}$, $\Delta_\mathsf{p}(t,e) \downarrow$ if and only if $e \not\in t$ and every event in $\mathsf{Events_p}$ whose replica coincide with e and has smaller identifier than e is included in t.

Given such a program P, the proof proceeds as follows:

1. There exists a finite trace t of $P \parallel I_E$ that contains no receive action (Lemma 5.7.5): Since I_E is available, it can always delay receiving messages, and execute other actions instead. Then, as P is a finite program, such an execution must be finite.
2. The trace t induces a history $h_\mathsf{v} = (E, \mathsf{so}, \mathsf{wr})$ and an abstract execution $\xi_\mathsf{v} = (h, \mathsf{rb}, \mathsf{ar})$ where $\mathsf{rb} = \mathsf{so}$ (ar is arbitrary as long as $\mathsf{rb} \subseteq \mathsf{ar}$). As I_E is valid w.r.t. Spec, ξ_v is valid w.r.t. Spec. Next, we prove that since $\mathsf{rb} = \mathsf{so}$, events in ξ_v read the latest value w.r.t. so written on their associated object in ξ_v (Lemma 5.7.6). In particular, we deduce that all traces of P without `receive` events induce the same history and therefore, the induced history does not change when the induced arbitration order changes.
3. Since ar is a total order, either $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$ or $(e^{x_1}_{d_\mathsf{v}-1}, e^{x_0}_{d_\mathsf{v}-1}) \in \mathsf{ar}$. W.l.o.g., assume that $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$. By Lemma 5.7.7, we deduce that $e_0^{x_0} \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(v)}, [\xi_\mathsf{v}, \mathsf{CMod}])$. The proof is explained in Figure 5.5: if $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$, then all events $e_i^{x_0}$ form a path in such way that $\mathsf{v}_{x_0}(e_0^{x_0}, \ldots e^{x_0}_{\mathsf{len}(v)})$ holds in ξ_v.
4. Since $e^{x_0}_{\mathsf{len}(v)}$ is the only event at r_1 that reads or writes x_0 and events in ξ_v read the latests values w.r.t. so in ξ_v, we deduce that $e^{x_0}_{\mathsf{len}(v)}$ reads x_0 from `init`. However, as $e_0^{x_0} \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(v)}, [\xi_\mathsf{v}, \mathsf{CMod}])$ and `init` precedes $e_0^{x_0}$ in arbitration order, we

deduce that $e^{x_0}_{\mathsf{len}(v)}$ does not read the latest value w.r.t. ar, i.e. $\mathsf{rspec}(e^{x_0}_{\mathsf{len}(v)}) \downarrow$ but $\mathsf{wr}^{-1}_{x_0}(e^{x_0}_{\mathsf{len}(v)}) \neq \{\max_{\mathsf{ar}} \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(v)}, [\xi_{\mathsf{v}}, \mathsf{CMod}])\}$. Therefore, ξ_{v} is not valid w.r.t. Spec (see Definition 5.4.2). This contradicts the hypothesis that I_E is an implementation of Spec. □

Lemma 5.7.4. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification s.t.* CMod *is in normal form w.r.t.* OpSpec*. For every visibility formula* $v \in \mathsf{CMod}$*, there exists an abstract execution valid w.r.t.* Spec*,* ξ*, an object* x *and events* $e_0, \dots e_{\mathsf{len}(v)}$ *s.t.* $\mathsf{rspec}(e_{\mathsf{len}(v)}) \downarrow$ *and* $v_x(e_0, \dots e_{\mathsf{len}(v)})$ *holds in* ξ*.*

Proof. Let $v \in \mathsf{CMod}$ be a visibility formula. As CMod is normal form w.r.t. OpSpec, v is non-vacuous; so $\mathsf{CMod} \not\equiv \mathsf{CMod} \setminus \{v\}$. Hence, there exists an abstract execution valid w.r.t. Spec, ξ, an object x and a read event r s.t. $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}]) \neq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod} \setminus \{v\}])$. As $\mathsf{CMod} \setminus \{v\} \preccurlyeq \mathsf{CMod}$, we conclude that there exists events $e_0, \dots e_{\mathsf{len}(v)}$ s.t. $r = e_{\mathsf{len}(v)}$ and $v_x(e_0, \dots e_{\mathsf{len}(v)})$ holds in ξ. □

Lemma 5.7.5. *For every available storage implementation,* I_E*, there exists finite reachable trace* $t \in \mathcal{T}_{P \| I_E}$ *s.t.*

1. *t does not contain any action a s.t.* $\mathtt{op}(a) = \mathtt{receive}$.

2. *for every event* $e \in \mathsf{Events}_{\mathsf{p}}$ *there exists exactly one action* $a \in t$ *s.t.* $\mathtt{ev}(a) = e$ *and,*

3. *for every two actions* $a, a' \in t$ *in the same replica, if* $\mathtt{ev}(a) \downarrow$*,* $\mathtt{ev}(a') \downarrow$ *and* $\mathtt{id}(\mathtt{ev}(a)) < \mathtt{id}(\mathtt{ev}(a'))$*, then* $a <_t a'$

Proof. Let I_E be an available storage implementation. We construct a sequence of traces $\{t^i\}_{i \in \mathbb{N}}$ s.t. for each $i \in \mathbb{N}$ (1) t^i does not contain any $\mathtt{receive}$ action, (2a) for every event $e \in \mathsf{Events}_{\mathsf{p}}$ s.t. $\mathtt{id}(e) \leq \mathtt{id}(\mathtt{last}_{\mathtt{rep}(e)}(\pi_1(t^i)))$ there is exactly one action $a \in t^i$ s.t. $\mathtt{ev}(a) = e$, (2b) for every event $e \in \mathsf{Events}_{\mathsf{p}}$ s.t. $\mathtt{id}(e) > \mathtt{id}(\mathtt{last}_{\mathtt{rep}(e)}(\pi_1(t^i)))$ there is no action $a \in t^i$ s.t. $\mathtt{ev}(a) = e$, and (3) for every two actions $a, a' \in t$, if $\mathtt{ev}(a) \downarrow$, $\mathtt{ev}(a') \downarrow$ and $\mathtt{id}(\mathtt{ev}(a)) < \mathtt{id}(\mathtt{ev}(a'))$, then $a <_{t^i} a'$.

Let $t^0 = \mathtt{init}_{P \| I_E}$ be the first trace of our sequence. Clearly, t^0 satisfy properties (1), (2a), (2b) and (3). Then, let $n \in \mathbb{N}$ and, assuming that the trace t^n satisfy properties (1), (2a), (2b) and (3), we define t^{n+1}. If for every replica r and every event $e \in \mathsf{Events}_{\mathsf{p}}$, $\Delta_{\mathsf{p}}(\pi_1(t^n), e) \uparrow$, we define $t^{n+1} = t^n$. If not, let r_n be a replica and $e_n \in \mathsf{Events}_{\mathsf{p}}$ be an event s.t. $\Delta_{\mathsf{p}}(\pi_1(t^n), e_n) \downarrow$. As I_E is available, there exists an action a_n s.t. $\mathtt{op}(a'_n) \neq \mathtt{receive}$ and $\Delta_{P \| I_E}(t^n, a_n) \downarrow$. We then define $t^{n+1} = \Delta_{P \| I_E}(t^n, a_n)$.

By induction hypothesis on t^n, t^n satisfies properties (1), (2a), (2b) and (3). We show that t^{n+1} also satisfies (1), (2a), (2b) and (3). Without loss of generality, we assume that $t^{n+1} \neq t^n$ as otherwise the result immediately holds. First, as t^n satisfies (1) and a_n is not a $\mathtt{receive}$ action, t^{n+1} satisfies property (1). Properties (2a) and (2b) immediately hold from the definition of $\Delta_{P \| I_E}$.

Finally, for proving that t^{n+1} satisfies (3), let $a, a' \in t^n$ be distinct actions s.t. $\mathtt{ev}(a) \downarrow$, $\mathtt{ev}(a') \downarrow$ and $\mathtt{id}(\mathtt{ev}(a)) < \mathtt{id}(\mathtt{ev}(a'))$. If $a, a' \neq a_n$, as t^n satisfies (3), $a <_{t^n} a'$ and therefore, $a <_{t^{n+1}} a'$. Otherwise, note that as t^n satisfies (2b), for every event $e \in \pi_1(t^n)$, $\mathtt{id}(e) \leq$

$\mathtt{id}(\mathtt{ev}(a_n))$. Moreover, as no two events in $\mathsf{Events_p}$ have identical identifier, traces do not contain the same event twice and $a \neq a'$, we deduce that $a' = a_n$. As $a_n = \mathtt{last}_{r_n}(t^{n+1})$, we conclude that $a <_{t^{n+1}} a'$.

By construction, t^∞ is a trace in $\mathcal{T}_{P\|I_E}$. As P is finite and I_E is available, every trace $t \in \mathcal{T}_{P\|I_E}$ is finite. We show by contradiction that there exists some $k \in \mathbb{N}$ s.t. $t^k = t^{k+1}$. Consider the sucession of traces $\{t^n\}_{n\in\mathbb{N}}$ and let us assume that $t^k \neq t^{k+1}$ for any $k \in \mathbb{N}$. In such case, we define t^∞ as the limit of such sucession, i.e., the trace obtained by executing events actions $a_i, 0 \leq i \leq \mathbb{N}$ (which are well-defined by construction). Such infinite trace belongs to $\mathcal{T}_{P\|I_E}$. However, as P is finite and I_E is available, every trace $t \in \mathcal{T}_{P\|I_E}$ is finite. Thus, t^∞ must be finite; which contradicts its construction. Hence, such k exists.

We show that the trace t^k is the searched trace. Clearly, as t^k satisfies (1) and (3), it suffices to prove that it also satisfies (2). On one hand, as $t^k = t^{k+1}$, for every event $e \in P$, $\Delta_{\mathsf{p}}(\pi_1(t^k), e) \uparrow$. Hence, for every replica $r_l, l \in \{0,1\}$, $\mathtt{last}_r(\pi_1(t^k)) = e^{x_{1-l}}_{\mathsf{len}(v)}$. By construction of $\mathsf{Events_p}$, every event $e \in \mathsf{Events_p}$ with replica r_l has smaller identifier than $e^{x_{1-l}}_{\mathsf{len}(v)}$. Therefore, as t^k satisfies (2a), there is exactly one action $a' \in t^k$ s.t. $\mathtt{ev}(e') = e$; so t^k satisfies (2).

□

Lemma 5.7.6. *For every pair of indices $i, -1 \leq i \leq \mathsf{len}(v)$, $l \in \{0,1\}$,*

- *If $e^{x_l}_i$ is a read event, then $(\{w^{x_l}_i\}, e^{x_l}_i) \in \mathsf{wr}_{\tilde{x}^{x_l}_i}$.*
- *If $e^{x_l}_i$ is a write event s.t. $\mathtt{wval}(e^{x_l}_i)(\tilde{x}^{x_l}_i) \downarrow$, then $\mathsf{wspec}(e^{x_l}_i)(\mathtt{wval}(w^{x_l}_i)(\tilde{x}^{x_l}_i)) \downarrow$.*

Proof. We prove the result by induction on i; where the base case, $i = -1$, trivially holds. For showing the inductive case, let us assume that the result holds for every event $e^{x_{l'}}_{i'}, -1 \leq i' < i, l' \in \{0,1\}$, and let us show it for events $e^{x_0}_i, e^{x_1}_i$. We divide the proof in two blocks, whether $e^{x_l}_i$ is a read event, and $e^{x_l}_i$ is a write event.

For the first part, we note that by construction of ξ_v using Lemma 5.7.5 we know that ξ_v does not contain any $\mathtt{receive}$ event, $\mathsf{rb} = \mathsf{so}$. Hence, as ξ_v is valid w.r.t. Spec, $\mathsf{wr} \subseteq \mathsf{rb} = \mathsf{so}$. Thus, $e^{x_l}_i$ reads $\tilde{x}^{x_l}_i$ from an event that precedes it in session order. In particular, by Definition 5.4.2, $\mathsf{wr}^{-1}_{\tilde{x}^{x_l}_i}(e^{x_l}_i) = \{\max_{\mathsf{ar}} \mathsf{ctxt}_{\tilde{x}^{x_l}_i}(e^{x_l}_i, [\xi_v, \mathsf{CMod}])\}$; so $\mathsf{wr}^{-1}_{\tilde{x}^{x_l}_i}(e^{x_l}_i) = \{w^{x_l}_i\}$.

For the second part, we can assume w.l.o.g. that $e^{x_l}_i$ is a conditional write, as otherwise the result immediately holds. By the choice of $e^{x_l}_i$, in this case, we conclude that $\mathsf{wspec}(e^{x_l}_i)(\mathtt{wval}(w^{x_l}_i)(\tilde{x}^{x_l}_i)) \downarrow$.

□

Lemma 5.7.7. *For every $l \in \{0,1\}$, if $(e^{x_l}_{d_v-1}, e^{x_{1-l}}_{d_v-1}) \in \mathsf{ar}$, then $e^{x_l}_0 \in \mathsf{ctxt}_{x_l}(e^{x_l}_{\mathsf{len}(v)}, [\xi_v, \mathsf{CMod}])$.*

Proof. For proving that $e^{x_l}_0 \in \mathsf{ctxt}_{x_l}(e^{x_l}_{\mathsf{len}(v)}, [\xi_v, \mathsf{CMod}])$, we show that $v_{x_l}(e^{x_l}_0, e^{x_l}_{\mathsf{len}(v)})$ holds in ξ_v. Observe that by the choice of events and Lemma 5.7.6 $e^{x_l}_0$ writes x_l in ξ_v and $\mathsf{wr}^{-1}_{x_l}(e^{x_l}_{\mathsf{len}(v)}) \neq \emptyset$ holds in ξ_v. Therefore, to conclude the result, we prove that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e^{x_l}_{i-1}, e^{x_l}_i) \in \mathsf{Rel}^v_i$.

For proving it, we observe that CMod is in simple form. Thus, for every $i, 1 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}^{\mathsf{v}}_i$ is either $\mathsf{so}, \mathsf{wr}, \mathsf{rb}$ or ar; which simplify our analysis. First, if $i = d_v$, by definition of d_v, $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{ar}$. By hypothesis, $(e^{x_l}_{d_v-1}, e^{x_{1-l}}_{d_v-1}) \in \mathsf{ar}$. In such case, as $\mathtt{id}(e^{x_{1-l}}_{d_v-1}) < \mathtt{id}(e^{x_l}_{d_v})$ and $\mathtt{rep}(e^{x_{1-l}}_{d_v-1}) = \mathtt{rep}(e^{x_l}_{d_v})$, $(e^{x_{1-l}}_{d_v-1}, e^{x_l}_{d_v}) \in \mathsf{so}$. Therefore, as $\mathsf{so} \subseteq \mathsf{ar}$ and ar is a transitive relation, we deduce that $(e^{x_l}_{d_v-1}, e^{x_l}_{d_v}) \in \mathsf{ar}$.

Next, if $i \neq d_v$, we notice that $(e^{x_0}_{i-1}, e^{x_0}_i) \in \mathsf{so} \subseteq \mathsf{rb} \subseteq \mathsf{ar}$. Hence, if $\mathsf{Rel}^{\mathsf{v}}_i$ is either so, rb or ar, the result immediately holds. Otherwise, if $\mathsf{Rel}^{v}_i = \mathsf{wr}$, we show that $e^{x_0}_i$ is a read event and $e^{x_0}_{i-1} = w^{x_0}_i$; which let us conclude that $(e^{x_0}_{i-1}, e^{x_0}_i) \in \mathsf{wr}$ thanks to Lemma 5.7.6.

First, we show that if $i \neq \mathsf{len}(v)$ and $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, then $w^{x_l}_i = e^{x_l}_{i-1}$. Thanks to the choice of P, if $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, then $e^{x_l}_i$ is a write event s.t. $\mathtt{wval}(e^{x_l}_{i-1})(\tilde{x}^{x_l}_i) \downarrow$. By Lemma 5.7.6, we deduce that $e^{x_l}_{i-1}$ writes $\tilde{x}^{x_l}_{i-1}$ in ξ_v. As $i \neq \mathsf{len}(v)$, $\tilde{x}^{x_l}_{i-1} = \tilde{x}^{x_l}_i$. Also, as $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, $\mathtt{rep}(e^{x_l}_i) = \mathtt{rep}(e^{x_l}_{i-1})$. Altogether, we deduce that $e^{x_l}_{i-1}$ is an event writing $\tilde{x}^{x_l}_i$ that is the immediate predecessor of $e^{x_l}_i$ w.r.t. so. Hence, $w^{x_l}_i = e^{x_l}_{i-1}$.

Finally, we show that $\mathsf{Rel}^{v}_{\mathsf{len}(v)} \neq \mathsf{wr}$ and conclude the result. We prove the contrapositive, that if $\mathsf{Rel}^{\mathsf{v}}_{\mathsf{len}(v)} = \mathsf{wr}$, v is vacuous w.r.t. Spec. If $\mathsf{Rel}^{\mathsf{v}}_{\mathsf{len}(v)} = \mathsf{wr}$, for every abstract execution ξ' valid w.r.t. Spec, object x and a collection of events $f_0, \ldots f_{\mathsf{len}(v)}$, if $v_x(f_0, \ldots, f_{\mathsf{len}(v)})$ holds in ξ', then $(f_{\mathsf{len}(v)-1}, f_{\mathsf{len}(v)}) \in \mathsf{wr}$. Thus, ξ' is valid w.r.t. $(\mathsf{CMod} \setminus \{v\}, \mathsf{OpSpec})$. Hence, v is vacuous w.r.t. OpSpec. $\square$

5.8 Generalized Distributed Storage Specifications

We describe a generalization of the basic storage specifications from Section 5.4 along three dimensions: a larger class of consistency models, multi-object operations, and more general read behaviors. To rule out anomalous behaviors in this generalization, we introduce a set of additional assumptions. Figure 5.6 summarizes the structure of storage specifications and the relationship between basic and generalized specifications in terms of assumptions.

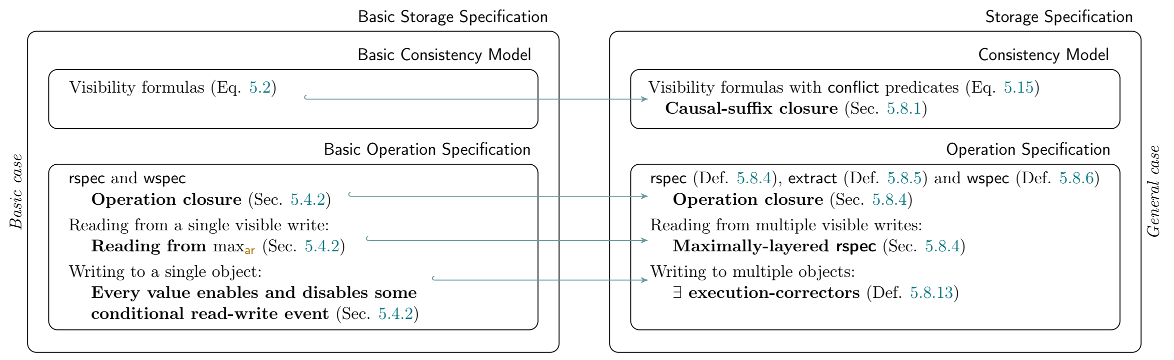
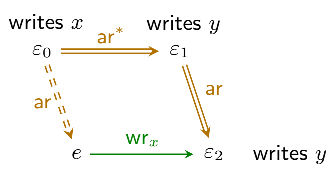

Basic Storage Specification
Basic Consistency Model
Visibility formulas (Eq. 5.2)
Basic Operation Specification
rspec and wspec
Operation closure (Sec. 5.4.2)
Reading from a single visible write:
Reading from max_ar (Sec. 5.4.2)
Writing to a single object:
Every value enables and disables some conditional read-write event (Sec. 5.4.2)
Basic case
Storage Specification
Consistency Model
Visibility formulas with conflict predicates (Eq. 5.15)
Causal-suffix closure (Sec. 5.8.1)
Operation Specification
rspec (Def. 5.8.4), extract (Def. 5.8.5) and wspec (Def. 5.8.6)
Operation closure (Sec. 5.8.4)
Reading from multiple visible writes:
Maximally-layered rspec (Sec. 5.8.4)
Writing to multiple objects:
∃ execution-correctors (Def. 5.8.13)
General case

Figure 5.6: Conceptual map relating basic and generalized storage specifications (Sections 5.4 and 5.8). Storage specifications are composed of consistency models and operation specifications. Assumptions are written in bold text. Arrows denote how definitions/assumptions translate from the basic case to the general case.

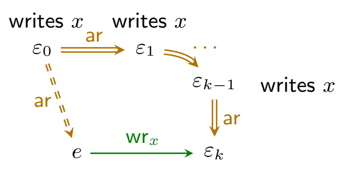
5.8.1 Consistency Models

The set of basic consistency models (Section 5.4.1) does *not* include (parallel) snapshot isolation, and the version of k-bounded staleness considered in Section 5.2. Snapshot Isolation, k-Bounded Staleness and Parallel Snapshot Isolation are defined, respectively, using the visibility formulas Conflict (Figure 5.7a), k-Bounded (Figure 5.7b)[5], and n-PSI (Figure 5.7c).



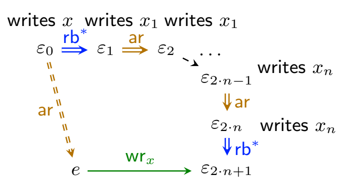
$\exists \varepsilon_1 . (\varepsilon_0, \varepsilon_1) \in \mathsf{ar}^* \land (\varepsilon_1, \varepsilon_2) \in \mathsf{ar} \land$
$\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \ \land \ \varepsilon_0$ writes $x \land$
$\mathsf{conflict}(\varepsilon_1, \varepsilon_2)$

(a) Conflict



$\exists \varepsilon_1 \dots \varepsilon_{k-1} . \ \bigwedge_{i=1}^{k} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{ar} \land$
$\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \land$
$\mathsf{conflict}_x(\varepsilon_0, \dots \varepsilon_{k-1})$

(b) k-Bounded



$\exists \varepsilon_1 \dots \varepsilon_{2n} . \bigwedge_{i=0}^{n} (\varepsilon_{2 \cdot i}, \varepsilon_{2 \cdot i+1}) \in \mathsf{rb}^* \land$
$\bigwedge_{i=1}^{n} (\varepsilon_{2 \cdot i-1}, \varepsilon_{2 \cdot i}) \in \mathsf{ar} \land$
$\mathsf{wr}_x^{-1}(\varepsilon_1) \neq \emptyset \land \ \varepsilon_0$ writes x
$\bigwedge_{i=1}^{n} \mathsf{conflict}(\varepsilon_{2 \cdot i-1}, \varepsilon_{2 \cdot i})$

(c) n-PSI

Figure 5.7: Conflict, k-Bounded and n-PSI visibility formulas used to define *Snapshot Isolation* (SI), *Bounded Staleness* (BS) and *Parallel Snapshot Isolation* (PSI). SI is defined by Prefix (Figure 5.4c) and Conflict, BS is defined by k-Bounded and Return-Value (Figure 5.4a), and PSI is defined by Causal (Figure 5.4b) and the set of visibility formulas $\{\text{n-PSI} \mid n \geq 1\}$.

To include such consistency models in our formalization, we extend the syntax of visibility formulas so that the intermediate events can be further constrained via the wrCons formula:

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_n) ::= \exists \varepsilon_1, \dots, \varepsilon_{n-1} . \bigwedge_{i=1}^{n} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^{\mathsf{v}} \land \mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset \land \mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0, \dots \varepsilon_n) \tag{5.15}$$

The formula $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0, \dots \varepsilon_n)$ is a conjunction of predicates $\mathsf{conflict}(E)$ and $\mathsf{conflict}_x(E)$ with $E \subseteq \{\varepsilon_0, \dots \varepsilon_n\}$. The predicate $\mathsf{conflict}(E)$ (resp., $\mathsf{conflict}_x(E)$) means that *all* the events in E write on some object y (resp., the object x). Since we want to preserve the constraint ε_0 writes x from basic visibility formulas, we require that there exists a set $E \subseteq \{\varepsilon_0, \dots \varepsilon_n\}$ s.t. $\varepsilon_0 \in E$ and $\mathsf{conflict}_x(E)$ is included in $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0, \dots \varepsilon_n)$ (E can be the singleton ε_0). The interpretation of a conflict predicate in an abstract execution ξ is done as expected: a predicate $\mathsf{conflict}(E)$ (resp., $\mathsf{conflict}_x(E)$) holds iff there exists an object y s.t. for every $e \in E$, e writes y in ξ (resp. e writes x in ξ). As before, the predicate ε writes y is true iff $\mathtt{wval}(e)(y) \downarrow$.

From this point on, a consistency model is defined as a set of visibility formulas, as in Equation (5.15).

Normal Form. We generalize the normal form of a consistency model to take into account conflict predicates. A consistency model in normal form only contains visibility formulas

[5]Our version of k-Bounded Staleness corresponds to the (k, T)-Bounded Staleness with $T = \infty$ as defined in [2].

that are simple, non-vacuous and "conflict-maximal". A *conflict-strengthening* of a visibility formula v is a visibility formula v' obtained from v by (1) replacing some occurrence of $\mathsf{conflict}(E)$ (resp., $\mathsf{conflict}_x(E)$) with $\mathsf{conflict}(E')$ (resp., $\mathsf{conflict}_x(E')$) where E' is a strict superset of E or (2) removing predicate $\mathsf{conflict}(E)$ if $\mathsf{conflict}_x(E)$ also belongs to v. A visibility formula v is *conflict-maximal* w.r.t. OpSpec iff there is no conflict-strengthening v' such that for every execution ξ over events in $\mathsf{Events}[\mathsf{OpSpec}]$, object x, and events $e_0, \ldots e_{\mathsf{len}(\mathsf{v})}$, if $\mathsf{v}_x(e_0, \ldots e_{\mathsf{len}(\mathsf{v})})$ holds in ξ, then $\mathsf{v}'_x(e_0, \ldots e_{\mathsf{len}(\mathsf{v})})$ holds in ξ as well. A consistency model CMod is *conflict-maximal* w.r.t. OpSpec iff all its visibility formulas are conflict-maximal w.r.t. OpSpec.

For example, if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, any instance of ε_i must write on some object y. In conflict-maximal visibility formulas, this fact is represented with a conflict predicate ($\mathsf{conflict}(E)$ or $\mathsf{conflict}_x(E)$) s.t. $\varepsilon_{i-1} \in E$. If OpSpec requires that every event reading y also writes on y, then in a conflict-maximal visibility formula, both $\varepsilon_{i-1}, \varepsilon_i$ belong to E. In general, if in any abstract execution, the events instantiating $\varepsilon_{i_1}, \ldots, \varepsilon_{i_j}$ from v_x always conflict (resp. they always write x), then the visibility formula v must contain the predicate $\mathsf{conflict}(\varepsilon_{i_1}, \ldots, \varepsilon_{i_j})$ (resp. $\mathsf{conflict}_x(\varepsilon_{i_1}, \ldots, \varepsilon_{i_j})$).

Definition 5.8.1. *A consistency model* CMod *is called in* normal form w.r.t. a operation specification OpSpec *if it contains only simple, conflict-maximal visibility formulas and no visibility formula from* CMod *is vacuous w.r.t.* OpSpec.

Under some operation specifications, consistency models can be equivalent due to conflict predicates. For example, in a storage with only `FAA` operations, `SI` and `SER` are equivalent due to the $\mathsf{Conflict}$ visibility formula: in this specification, every event is both a read and a write event and so any event reading x conflicts with an event writing x.

Similarly to Section 5.6, we say that a consistency model CMod is *arbitration-free* w.r.t. an operation specification OpSpec if there exists a consistency model in general normal form w.r.t. OpSpec that is equivalent to CMod and whose visibility formulas are arbitration-free. Section 5.11 demonstrates the existence of a normal form and shows that it is not possible for two normal forms to differ solely in that one includes only arbitration-free visibility formulas while the other does not. This result confirms that arbitration-freedom is not a property of the chosen normal form, but rather an inherent characteristic of the definitions of CMod and OpSpec.

Causal Suffix Closure. We introduce an assumption about consistency models which is used in the proof of the AFC theorem in order to find counterexamples to availability that involve only two replicas. This assumption is satisfied by all practical cases that we are aware of (see Example 5.8.3).

Therefore, we assume that every normal form CMod of a consistency model is *closed under causal suffixes*, i.e., for every visibility formula $\mathsf{v}_x \in \mathsf{CMod}$, CMod contains every arbitration-free "suffix" of v_x that starts with an event writing x. Thinking about a visibility formula v as a path of dependencies (between the pairs $(\varepsilon_{i-1}, \varepsilon_i)$), a suffix of v is a suffix of that path. For example, the visibility formulas s and s' described in Equation (5.17) and Equation (5.18) are suffixes of the visibility formula in Equation (5.16).

$$\mathsf{v}_x(\varepsilon_0, \varepsilon_3) = \exists \varepsilon_1, \varepsilon_2.(\varepsilon_0, \varepsilon_1) \in \mathsf{rb} \wedge (\varepsilon_1, \varepsilon_2) \in \mathsf{ar} \wedge (\varepsilon_2, \varepsilon_3) \in \mathsf{so} \wedge \\ \mathsf{wr}_x^{-1}(\varepsilon_3) \neq \emptyset \wedge \mathsf{conflict}_x(\varepsilon_0, \varepsilon_1, \varepsilon_2) \tag{5.16}$$

$$s_x(\varepsilon_1, \varepsilon_3) = \exists \varepsilon_2.(\varepsilon_1, \varepsilon_2) \in \mathsf{ar} \ \wedge \ (\varepsilon_2, \varepsilon_3) \in \mathsf{so} \ \wedge \ \mathsf{wr}_x^{-1}(\varepsilon_3) \neq \emptyset \wedge \ \mathsf{conflict}_x(\varepsilon_1, \varepsilon_2) \tag{5.17}$$

$$s'_x(\varepsilon_2, \varepsilon_3) = (\varepsilon_2, \varepsilon_3) \in \mathsf{so} \ \wedge \ \mathsf{wr}_x^{-1}(\varepsilon_3) \neq \emptyset \wedge \ \mathsf{conflict}_x(\varepsilon_2) \tag{5.18}$$

Formally, let v_x be a visibility formula defined as in Equation (5.15). Let $\mathsf{conflict}_x(\mathsf{v}_x)$ be the union of the sets E such that $\mathsf{conflict}_x(E)$ occurs in the definition of v_x. For any variable $\varepsilon_k \in \mathsf{conflict}_x(\mathsf{v}_x)$, the ε_k-suffix of v_x is the formula obtained by (1) removing the quantifiers for the first k quantified events, $e_1 \dots e_k$, and (2) removing all occurrences of the (now) free variables $e_0, \dots e_{k-1}$, i.e.:

$$\mathsf{suff}_x(\mathsf{v}_x, k)(\varepsilon_k, \varepsilon_n) ::= \exists \varepsilon_{k+1}, \dots, \varepsilon_{n-1}. \bigwedge_{i=k+1}^{n} (\varepsilon_{i-1}, \varepsilon_i) \in \mathsf{Rel}_i^{\mathsf{v}} \\ \wedge \mathsf{wr}_x^{-1}(\varepsilon_n) \neq \emptyset \wedge \mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_k, \dots \varepsilon_n)$$

where $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_k, \dots \varepsilon_n)$ is obtained from $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0, \dots \varepsilon_n)$ by projecting all the conflict predicates over the set of events $E_k = \{\varepsilon_k, \dots, \varepsilon_n\}$, i.e., a predicate $\mathsf{conflict}(E)$ (resp. $\mathsf{conflict}_x(E)$) occurs in $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_0, \dots \varepsilon_n)$ iff $\mathsf{conflict}(E \cap E_k)$ (resp. $\mathsf{conflict}_x(E \cap E_k)$) occurs in $\mathsf{wrCons}_x^{\mathsf{v}}(\varepsilon_k, \dots \varepsilon_n)$.

We refer to arbitration-free suffixes as *causal*, since the remaining dependencies intuitively reflect broader notions of causality. The intuition behind this notion of closure is that the context of an invocation should be upward-closed with respect to causality—meaning that if an update (writing x) is included, then any later updates (writing x) along the dependency path defined by the visibility formula that lie in its causal past must also be included.

We say that a visibility formula v' *subsumes* a visibility formula v of the same length if for every $i, 1 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{v'}$ is stronger or equal than $\mathsf{Rel}_i^{\mathsf{v}}$. We say that rb is stronger than so and wr, and ar is stronger than rb, so and wr. The extension of "being stronger" to any relation Rel described using Equation (5.3) is done as expected, as all our operators are positive (there are no negations).

Definition 5.8.2. *A consistency model* CMod *is* closed under causal suffixes *if for every* $\mathsf{v}_x \in \mathsf{CMod}$ *and* $\varepsilon_k \in \mathsf{conflict}_x(\mathsf{v}_x)$*,* CMod *includes some visibility formula* v' *that subsumes every arbitration-free suffix of* v*.*

Example 5.8.3. *A consistency model containing the visibility formula* v *in Equation* (5.16) *must also contain the visibility formula* s' *in order to be closed under causal suffixes. Note that* s *uses arbitration and it is not required to be included.*

Any basic consistency model is closed under causal suffixes because every basic visibility formula has no proper arbitration-free suffix. Indeed, $\mathsf{conflict}_x(\mathsf{v}_x)$ *contains just the first event* ε_0 *(assuming that* ε_0 writes x *is rewritten as* $\mathsf{conflict}_x(\{\varepsilon_0\})$*). The models described in Figures 5.4 and 5.7 are trivially closed under causal suffixes because their visibility formulas have no arbitration-free suffixes.*

5.8.2 Operation Specifications

We generalize operation specifications to allow operations to access (read or write) multiple objects, and to support read values that are not limited to the inputs of individual write operations. For example, this includes multi-value reads that return all concurrently written values for an object, or counter reads that return an aggregated value computed from all observed increments.

The generalized reading behavior is modeled using two functions rspec and extract described hereafter. We also introduce a generalized wspec function. Therefore, rspec selects from a given context the events (updates) which are relevant for a reading invocation, extract defines the value read by an invocation, if any (based on the output of rspec), and wspec defines the value written by an invocation, if any (to model conditional read-writes, this is based on the output of extract).

Definition 5.8.4. *A* read specification $\mathsf{rspec} : \mathsf{Events} \to \mathsf{Keys} \to \mathsf{Contexts} \to \mathcal{P}(\mathsf{Events})$ *is a function such that for every object x, context $c = (E, \mathsf{rb}, \mathsf{ar})$ and event e:*

1. *well-formedness:* $\mathsf{rspec}(e)(x, c) \subseteq E$*, and if e is an initial event,* $\mathsf{rspec}(e)(x, c) = \emptyset$*, and*
2. *unconditional reading: if* $\mathsf{rspec}(e)(x, c) \neq \emptyset$ *for some context c, then for every non-empty context c',* $\mathsf{rspec}(e)(x, c') \neq \emptyset$

Equations (5.19) to (5.21) describe the read specifications of faacas, a key-value store k-mv with $\mathtt{PUT}(x, v)$ and multi-value $\mathtt{GET}(x)$ operations (Section 5.9.2), and a collection of distributed counters counter with $\mathtt{inc}(x)$ and $\mathtt{rd}(x)$ operations (Section 5.9.3). Concerning the relationship to *basic* read specifications, note that the faacas specification in Equation (5.4) was simpler because the constraint from Equation (5.19) was imposed in the notion of validity for abstraction executions (Definition 5.4.2). For multi-value reads (Equation (5.20)), the read specification selects the maximal elements in the receive-before relation (which models causality), and for a counter (Equation (5.21)), it returns all events in the context.

$$\mathsf{rspec}(r)(x, c) = \begin{cases} \{\max_{\mathsf{ar}} E\}, & \text{if } r \in \{\mathtt{GET}(x), \mathtt{FAA}(x, v), \mathtt{CAS}(x, v, v')\} \text{ and } c = (E, \mathsf{rb}, \mathsf{ar}) \\ \emptyset, & \text{otherwise} \end{cases} \tag{5.19}$$

$$\mathsf{rspec}(r)(x, c) = \begin{cases} \max_{\mathsf{rb}} E, & \text{if } r = \mathtt{GET}(x) \text{ and } c = (E, \mathsf{rb}, \mathsf{ar}) \\ \emptyset, & \text{otherwise} \end{cases} \tag{5.20}$$

$$\mathsf{rspec}(r)(x, c) = \begin{cases} E, & \text{if } r = \mathtt{rd}(x) \text{ and } c = (E, \mathsf{rb}, \mathsf{ar}) \\ \emptyset, & \text{otherwise} \end{cases} \tag{5.21}$$

The extract specification below computes the value returned from an object x based on the set of invocations writing x returned by the read specification which are paired with values they write (this will become clearer when defining the application of these functions on an abstract execution).

Definition 5.8.5. *An* extract specification $\mathsf{extract} : \mathsf{Events} \to \mathsf{Keys} \to \mathcal{P}(\mathsf{Events} \times \mathsf{Vals}) \to \mathsf{Vals}$*, such that* $\mathsf{extract}(\mathtt{init})$ *is defined for every initial event* $\mathtt{init}$*.*

Equation (5.22) describes the extract specification of faacas: the value extracted for `GET`, `FAA` and `CAS` coincides with the value written by some previous `PUT`/`FAA`/`CAS` operation. Equation (5.23) describes the extract specification of $\mathsf{k\text{-}mv}$: the value extracted for `GET` is the set of values written by some previous `PUT`. In the case of $\mathsf{counter}$, Equation (5.24), the value extracted for `rd` returns the number of increment invocations in the input, which equals $|R|$ minus one for the initial event `init` which is always included in R (since it is so before all other events).

$$\mathsf{extract}(r)(x,R) = \begin{cases} v & \text{if } r \in \{\mathtt{GET}(x), \mathtt{FAA}(x,v'), \mathtt{CAS}(x,v',v'')\} \\ & \text{ and } R = \{(w,v)\} \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.22}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} \{v \mid (\_,v) \in R\} & \text{if } r = \mathtt{GET}(x) \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.23}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} |R|-1 & \text{if } r = \mathtt{rd}(x) \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.24}$$

Finally, the write specification computes the value written by an invocation to an object x, based on the values it reads. This makes it possible to model atomic read-writes, e.g., a compare-and-swap, which may write or not depending on what they read, or the value they write may change depending on what they read, e.g., a Fetch-and-Add.

Definition 5.8.6. *A* write specification $\mathsf{wspec} : \mathsf{Events} \to \mathsf{Keys} \to \mathsf{Vals} \to \mathsf{Vals}$ *is a function such that* $\mathsf{wspec}(\mathtt{init})$ *is defined for every initial event* `init`*.*

The write specification of faacas and $\mathsf{k\text{-}mv}$, Equation (5.25), describes that its write operations are `PUT`, `FAA` and `CAS`. `PUT` and `FAA` unconditionally writes on x while `CAS` does it depending on the read-and-extracted value of x; where x is the only object accessed by the invocation. In the case of $\mathsf{counter}$, Equation (5.27), only the operation $\mathtt{inc}(x)$ writes, writing a dummy value 1 just to indicate that the write has taken place.

$$\mathsf{wspec}(w)(x,v) = \begin{cases} v' & \text{if } w = \mathtt{PUT}(x,v') \\ v+v' & \text{if } w = \mathtt{FAA}(x,v') \\ v'' & \text{if } w = \mathtt{CAS}(x,v',v'') \land v = v' \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.25}$$

$$\mathsf{wspec}(w)(x,v) = \begin{cases} v' & \text{if } w = \mathtt{PUT}(x,v') \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.26}$$

$$\mathsf{wspec}(w)(x,\_) = \begin{cases} 1 & \text{if } w = \mathtt{inc}(x) \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.27}$$

Definition 5.8.7. *An* operation specification *is a tuple* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *where* E *is a set of events.* $\mathsf{Events}[\mathsf{OpSpec}]$ *refers to the set of events* E *in an operation specification.*

Section 5.9 contains more examples of operation specifications, including SQL statements.

5.8.3 Validity w.r.t. Storage Specifications

We extend the notion of validity for abstract executions to (general) storage specifications, in a way that is similar to the case of basic storage specifications (Section 5.4.3).

We use the extension of rspec, extract, and wspec to abstract executions defined below:

$$\begin{aligned}
&\mathsf{rspec}(e)(x, [\xi, \mathsf{CMod}]) = \mathsf{rspec}(e)(x, \mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}]))\\
&\mathsf{extract}(e)(x, [\xi, \mathsf{CMod}]) = \mathsf{extract}(e)\left(x, \left\{(e', \mathtt{wval}(e)(x)) \,\middle|\, e' \in \mathsf{rspec}(e)(x, [\xi, \mathsf{CMod}])\right\}\right)\\
&\mathsf{wspec}(e)(x, [\xi, \mathsf{CMod}]) = \mathsf{wspec}(e)(x, \mathsf{extract}(e)(x, [\xi, \mathsf{CMod}]))
\end{aligned}$$

Definition 5.8.8. *Let* Spec = (CMod, OpSpec) *be a storage specification. An abstract execution* $\xi = (h, \mathsf{rb}, \mathsf{ar})$ *of a history* $h = (E, \mathsf{so}, \mathsf{wr})$ *is* valid *w.r.t.* Spec *iff*

- *ξ contains events from the operation specification, i.e.,* $E \subseteq \mathsf{Events}[\mathsf{OpSpec}]$*,*
- *for every event* $r \in E$*,* $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}])$*, and*
- *the value written by each event* $e \in E$ *to object* x *is consistent with* wspec*, i.e.,* $\mathtt{wval}(e)(x) = \mathsf{wspec}(e)(x, [\xi, \mathsf{CMod}])$*.*

A history h *is valid w.r.t.* Spec *iff there exists an abstract execution of* h *which is valid w.r.t.* Spec*.*

Observe that Definition 5.8.8 coincides with Definition 5.4.2 for storage systems that also admit basic storage specifications, e.g., faacas.

5.8.4 Assumptions About Operation Specifications

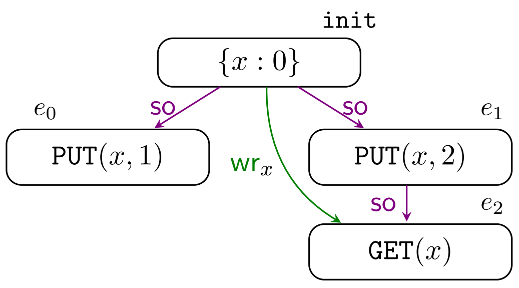
To avoid pathological behaviors in the generalization of specifications, we make several assumptions.

Maximally-Layered Read Specifications. For any basic operation specification OpSpec, the validity of an abstract execution w.r.t. a stronger consistency model (and OpSpec) implies validity w.r.t. a weaker one (see Lemma 5.6.5). In general, this is not true for operation specifications as described in this section (see Example 5.8.9). Therefore, we introduce an assumption about read specifications, called *maximally-layered*, which ensures that this property remains true.

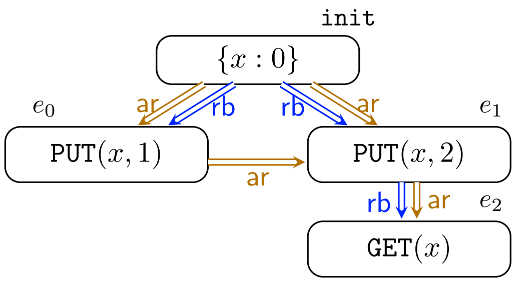
Example 5.8.9. *Let* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *be an operation specification of a key-value store with* GET *and* PUT *operations whose read specification is given by Equation (5.28).*

$$\mathsf{rspec}(e)(x, c) = \begin{cases} \{\max_{\mathsf{ar}} E\} & \text{if } \nexists e' \in E \text{ s.t. } \mathtt{rep}(e) \neq \mathtt{rep}(e') \text{ and } c = (E, \mathsf{rb}, \mathsf{ar}) \\ \mathtt{init} & \text{otherwise} \end{cases} \tag{5.28}$$

We compare the validity of the abstract execution ξ depicted in Figure 5.8 w.r.t. SC *and* CC *(observe that* $\mathtt{CC} \preccurlyeq \mathtt{SC}$*). Under* SC *both* e_0 *and* e_1 *are visible to* e_2*, which implies* $\mathsf{rspec}(e_2)(x, [\xi, \mathtt{SC}]) = \{\mathtt{init}\}$*. Therefore, ξ is valid w.r.t.* SC*. However, under* CC*, only* e_1 *is visible to* e_2*, which implies* $\mathsf{rspec}(e_2)(x, [\xi, \mathtt{CC}]) = \{e_1\}$*, and therefore, ξ is not valid w.r.t.* CC*.*



(a) History of a key-value store with PUT and GET.



(b) An abstract execution of the history in Figure 5.8a.

Figure 5.8: A history and an abstract execution of the operation specification in Example 5.8.9. For readability, we omit the so and wr relations from the abstract execution. Events e_1 and e_2 are executed in the same replica, different from e_0's replica.

Let $\leq$ be a partial order over a set A. A chain of $\leq$ is a subset of A which is totally ordered w.r.t. $\leq$. The *layer* of an element $a \in A$ is the size of the largest chain of $\leq$ which includes a but no elements smaller than a, and a maximal element. For instance, the layer of a maximal element is 1 (the aforementioned largest chain includes just the element itself), the level of a strict predecessor of a maximal element is 2, and so on. A subset $B \subseteq A$ is called *k-maximally layered* w.r.t. $\leq$ if B is the set of all elements in A of layer $k' \leq k$. When $\leq$ is also a total order, the notion of maximally layered is equivalent to being upward closed w.r.t. $\leq$. Otherwise, it is equivalent to being upward closed w.r.t. every total extension of $\leq$.

A read specification rspec is *k-maximally layered* w.r.t. ar (resp. rb^+) if for every object x, context $c = \{E, \mathsf{rb}, \mathsf{ar}\}$, and event e, either $\mathsf{rspec}(e)(x, c) = \emptyset$ or $\mathsf{rspec}(e)(x, c)$ is k-maximally layered w.r.t. ar (resp. rb^+). The *layer bound* of rspec is defined as k. To cover cases where there is no such k, we say that a read specification rspec is *∞-maximally layered* if for every x, context $c = \{E, \mathsf{rb}, \mathsf{ar}\}$, and event e, either $\mathsf{rspec}(e)(x, c) = \emptyset$ or E; and we say that the layer bound is ∞. When the layer bound and the partial order (ar or rb^+) are irrelevant, we simply say that rspec is *maximally layered.*

Example 5.8.10. *For example,* faacas *is* 1*-maximally layered w.r.t.* ar*,* k-mv *is* 1*-maximally layered w.r.t.* rb^+ *and* counter *is* ∞*-maximally layered. On the other hand, the read specification in Example 5.8.9 is not maximally layered since it can sometimes return* init *from a non-empty context.*

Lemma 5.8.11. *Let* OpSpec *be a maximall-layered operation specification and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be a pair of consistency models such that* CMod_2 *is stronger than* CMod_1*. Any abstract execution valid w.r.t.* $(\mathsf{CMod}_2, \mathsf{OpSpec})$ *is also valid w.r.t.* $(\mathsf{CMod}_1, \mathsf{OpSpec})$*.*

Operation Closure. As in Section 5.4.2, we assume that OpSpec contains at least a read and a write event. Also, we assume that all objects support a common set of operations with identical read and write behavior, and that these operations can be executed at any replica. Formally, for every event $e \in E$, replica r, and identifier id, there exists an event e' s.t. $\mathtt{rep}(e') = \mathsf{r}$, $\mathtt{id}(e') = \mathsf{id}$, $\mathsf{obj}(e') = \mathsf{obj}(e)$, $\mathsf{rspec}(e') = \mathsf{rspec}(e)$, $\mathsf{extract}(e') = \mathsf{extract}(e)$, and $\mathsf{wspec}(e') = \mathsf{wspec}(e)$.

We also assume that operations apply uniformly to any set of objects. To formalize this assumption, we define a notion of *domain* for an operation specification OpSpec which is any set of objects D s.t. there is an event $e \in \mathsf{Events}[\mathsf{OpSpec}]$ for which $\mathsf{obj}(e) = D$. We assume that domains are "symmetric", i.e. if D is a domain for OpSpec, then for every pair of objects $x \in D$ and $y \in \mathsf{Keys} \setminus D$, the set $D' = D \setminus \{x\} \cup \{y\}$ is also a domain for OpSpec. If OpSpec allows single-object read/write/read-write events (defined as in Section 5.4.2), we assume that for every object x, there exists a read/write/read-write event whose domain is $\{x\}$. Also, we assume that if OpSpec allows a *multi-object* read/write/read-write event e such that $\mathsf{obj}(e)$ is a finite set of size at least 2, then for every non-empty finite set $D \subseteq \mathsf{Keys}$, D is a domain of a read/write/read-write event in OpSpec.

Correctors. In addition, we assume that if OpSpec permits conditional read-write events–which write to a set of objects X based on values they read (possibly from other objects) in some context–then any execution can be extended with some conditional read-write event e that writes to every object in X, modulo a so-called correction defined below. This property is only relevant for events with $|\mathsf{obj}(e)| > 1$ (and therefore, irrelevant for basic storage specifications). Our proof will rely on the existence of such extensions.

Example 5.8.12. *To provide some intuition about the need for corrections, consider a specification formed of prefix consistency (*$\mathtt{PC}$*) and an operation specification with two multi-object operations,* $\mathtt{InsAbs}$ *and* $\mathtt{DelPre}$*, under Last-Writer-Wins (LWW) conflict resolution (i.e., the read specification selects the maximal invocation from the context w.r.t.* ar*) (see Section 5.9.4).* $\mathtt{InsAbs}(X, v)$ *checks for every object* $x \in X$ *if it is present, and inserts it with value* v *if not, and* $\mathtt{DelPre}(X)$ *deletes every object* $x \in X$ *as long as it was present.Assume an abstract execution* ξ*, and an event* e *from* ξ *whose context implies that* x *is absent and* y *is present. If* e *is an invocation of* $\mathtt{InsAbs}$ *(resp.,* $\mathtt{DelPre}$*), then it can not write both objects since* x *is absent and* y *is present.*

We introduce the notion of *corrector*, a set of auxiliary events that modify the context, ensuring the existence of an event that can write to both objects. For instance, in the scenario presented in Example 5.8.12, if e is an invocation of $\mathtt{InsAbs}(\{x, y\}, 1)$, the corrector will add a $\mathtt{DelPre}(\{y\})$ invocation in its context, so both objects are absent.

We start by defining some notations. Let $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ be a storage specification, $\xi = (h, \mathsf{rb}, \mathsf{ar})$ an abstract execution of a history $h = (E, \mathsf{so}, \mathsf{wr})$, and $e \in E$ an event. A *correction* of e in ξ with an event a, denoted by $\xi \overset{a}{\curlyvee} e$, is an abstract execution $\xi' = (h', \mathsf{rb}', \mathsf{ar}')$ associated to a history $h' = (E \cup \{a\}, \mathsf{so}', \mathsf{wr}')$ obtained by adding a as the immediate rb-predecessor and ar-predecessor of e. If $\mathtt{rep}(e) = \mathtt{rep}(a)$, then a is also the immediate so-predecessor of e. The write-read dependencies (wr^{-1}) of every event in ξ remain the same. Multiple corrections exist because the write-read and receive-before dependencies of a are not constrained. This allows flexibility on correcting ξ while preserving validity w.r.t. Spec.

The correction of ξ with a sequence of events $\vec{\mathsf{s}} = (a_1, a_2, \ldots)$, denoted by $\xi \overset{\vec{\mathsf{s}}}{\curlyvee} e$, is defined as expected, by iteratively correcting ξ with all events in $\vec{\mathsf{s}}$ in the order defined by $\vec{\mathsf{s}}$. Therefore, if e' is the immediate ar-predecessor of e in ξ, the ar order in $\xi \overset{\vec{\mathsf{s}}}{\curlyvee} e$ will have $a_1, a_2, \ldots$ inserted in between e' and e (in this order). Similarly for rb and possibly for so.

For a (partial) mapping $f : A \to B$ and a total order $<$ over A, the sequence of elements in B mapped by f and ordered according to $<$ is denoted by $\mathsf{seq}_<(f)$. Formally, $\mathsf{seq}_<(f) = (f(a_1), f(a_2), \ldots)$ such that $f(a_i)\downarrow$ and $a_i < a_{i+1}$ for all i. We omit the subscript $<$ when it is understood from the context.

Also, if ξ is an abstract execution, then $\xi \oplus e$ is an abstract execution obtained from ξ by appending e to ξ as the last event w.r.t. ar.

Corrector Assumption. If OpSpec allows conditional read-writes, then we assume that for every domain D, $W \subseteq D$, $x \in \mathsf{Keys}$ s.t. $x \in W$ if $W \neq \emptyset$, and abstract execution ξ, there exists

1. a conditional read-write e with $\mathsf{obj}(e) = D$ which is not contained in ξ, and
2. a partial mapping $a : D \setminus \{x\} \to \mathsf{Events}$ called *execution-corrector* for event e in an abstract execution $\xi \oplus e$.

We define execution-correctors as follows.

Definition 5.8.13. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification,* ξ *an abstract execution,* e *a conditional read-write event from* ξ *with* $\mathsf{obj}(e) = D$, $W \subseteq D$ *a set of objects, and* $x \in D$ *an object s.t.* $x \in W$ *if* $W \neq \emptyset$. *Also, let* $<$ *be a fixed total order on the set of objects. An* execution-corrector *for* (e, W, x, ξ) *is a partial mapping* $a : D \setminus \{x\} \to \mathsf{Events}$ *such that if*

$$\xi' = \xi \overset{\mathsf{seq}(a)}{\curlyvee} e \text{ and } \xi' \upharpoonright y = (\xi \overset{\mathsf{seq}(a \upharpoonright y)}{\curlyvee} e) \setminus \{e\} \text{ where } a \upharpoonright y = a \upharpoonright_{\{z \in \mathsf{Dom}(a) \mid z \leq y\}},$$

then the following hold:

1. *for every* $y \in D \setminus \{x\}$, *if* $a(y)$ *is defined and the correction up to* $a(y)$ *is valid w.r.t.* Spec, *then* $a(y)$ *writes only* y *in the correction: if* $a(y)\downarrow$ *and* $\xi' \upharpoonright y$ *is valid w.r.t.* Spec, *then for every object* $z \in \mathsf{Keys}$, $\mathsf{wspec}(a(y))(z, [\xi' \upharpoonright y, \mathsf{CMod}])\downarrow$ *iff* $z = y$, *and*
2. *for every* $y \in D$, *if the correction is valid w.r.t.* Spec, *then* e *reads* y *and additionally,* e *writes* y *only if* $y \in W$, *i.e.,* $\mathsf{rspec}(e)(y, [\xi', \mathsf{CMod}]) \neq \emptyset$ *and* $\mathsf{wspec}(e)(y, [\xi', \mathsf{CMod}])\downarrow$ *iff* $y \in W$.

Example 5.8.14. *We illustrate execution-correctors for the storage specification presented in Example 5.8.12, with* `InsAbs` *and* `DelPre` *as operations and* `PC` *as consistency model.*

Let ξ *be an abstract execution,* e *a* `DelPre`(D) *event from* ξ, $W \subseteq D$ *a non-empty set of objects and* $x \in W$. *For every object* y, *let* w_y *be the last event from the "read" context of* e *w.r.t.* `PC` *which writes* y *(by read context we mean the set of events selected by* rspec *from the context). In the following we assume that* w_x *is an insert event. Note that if* w_x *is a delete event, then there exists no execution-corrector for e (intuitively, the correction concerns objects different from* x, *and* `DelPre`(D) *will not delete an object which is already deleted).*

An execution-corrector for (e, W, x, ξ) *is the mapping* $a : D \setminus \{x\} \to \mathsf{Events}$ *defined below. The mapping* a *observes the update on* y *made by* w_y, *and overwrites it when necessary. Thus,* *when e reads y, y is present iff $y \in W$.*

$$a(y) = \begin{cases} \texttt{InsAbs}(\{y\}, v) & \textit{if } y \in W \textit{ and } w_y \textit{ deletes } y \textit{ in } \xi \\ \texttt{DelPre}(\{y\}) & \textit{if } y \notin W \textit{ and } w_y \textit{ inserts } y \textit{ in } \xi \\ \mathsf{undefined} & \textit{otherwise} \end{cases} \tag{5.29}$$

Observe that requiring that a is defined for all objects in D is too strict: if the read specification has a layer-bound of 1 and the events read a single object (as faacas), any correction will change the entire context read by e.

5.9 Examples of Operation Specifications

We present a list of well-know operation specifications and show that the satisfy the assumptions described in Section 5.8.4.

5.9.1 Key-Value Store with Fetch-And-Add and Compare-And-Swap Operations

The Key-Value Store with Fetch-And-Add and Compare-And-Swap (faacas) is an operation specification with four operations, $\texttt{PUT}(x, v)$, that puts value v to object x, $\texttt{GET}(x)$ that reads object x, $\texttt{FAA}(x, v)$ that reads the value v' of object x and writes $v' + v$, and $\texttt{CAS}(x, v, v')$, that reads x and writes v' iff the value read is v.

The following equations, corresponding to Equations (5.19), (5.22) and (5.25), describe the operation specification of faacas.

$$\mathsf{rspec}(r)(x, c) = \begin{cases} \{\max_{\mathsf{ar}} E\} & \text{if } r \in \{\texttt{GET}(x), \texttt{FAA}(x, v), \texttt{CAS}(x, v', v'')\} \\ & \ \text{and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{5.30}$$

$$\mathsf{extract}(r)(x, R) = \begin{cases} v & \text{if } r \in \{\texttt{GET}(x), \texttt{FAA}(x, v), \texttt{CAS}(x, v, v')\} \\ & \ \text{and } R = \{(w, v)\} \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.31}$$

$$\mathsf{wspec}(w)(x, v) = \begin{cases} v' & \text{if } w = \texttt{PUT}(x, v') \\ v + v' & \text{if } w = \texttt{FAA}(x, v') \\ v'' & \text{if } w = \texttt{CAS}(x, v', v'') \wedge v = v' \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.32}$$

The faacas is maximally layered w.r.t. ar, with 1 as its layer bound. As CAS is a single-object conditional read-write operation, it trivially allows execution-correctors.

5.9.2 Key-Value Multi-Value Store

The Key-Value Multi-Value Store (k-mv) [41, 19] is an operation specification with two operations, $\texttt{GET}(x)$, reading multiple concurrent values on a single object x, and $\texttt{PUT}(x, v)$, writing on a single object x the value v.

The following equations, corresponding to Equations (5.20), (5.22) and (5.25), describe the operation specification of k-mv.

$$\mathsf{rspec}(r)(x, c) = \begin{cases} \{\max_{\mathsf{rb}} E\} & \text{if } r = \texttt{GET}(x) \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{5.33}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} \{v \mid (\_, v) \in R\} & \text{if } r = \mathtt{GET}(x) \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.34}$$

$$\mathsf{wspec}(w)(x,\_) = \begin{cases} v & \text{if } w = \mathtt{PUT}(x,v) \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.35}$$

The k-v is maximally layered w.r.t. rb^+, with 1 as its layer bound.

5.9.3 Distributed Counter

The distributed counter (counter) [41] is an operation specification with two operations, $\mathtt{inc}(x)$, incrementing the value of x by 1, and $\mathtt{rd}(x)$, reading the amount of increments of x.

The following equations, corresponding to Equations (5.21), (5.24) and (5.27), describe the operation specification of counter.

$$\mathsf{rspec}(r)(x,c) = \begin{cases} E & \text{if } r = \mathtt{rd}(x) \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{5.36}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} |R| - 1 & \text{if } r = \mathtt{rd}(x) \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.37}$$

$$\mathsf{wspec}(w)(x,\_) = \begin{cases} 1 & \text{if } w = \mathtt{inc}(x) \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.38}$$

The counter is maximally layered w.r.t. ar, with ∞ as its layer bound.

5.9.4 Insert/Delete Last-Write-Wins

The Insert/Delete Last-Write-Wins (ins/del) is an operation specification with two multi-object operations. $\mathtt{InsAbs}(X, v)$ checks for every object $x \in X$ if it is present, and inserts it with value v if not, and $\mathtt{DelPre}(X)$ deletes every object $x \in X$ as long as it was already present.

Its operation specification is described as follows:

$$\mathsf{rspec}(r)(x,c) = \begin{cases} \{\max_{\mathsf{ar}} E\} & \text{if } r \in \left\{ \mathtt{InsAbs}(X,v), \mathtt{DelPre}(X) \right\}, \\ & \quad x \in X \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{5.39}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} v & \text{if } w \in \{\mathtt{InsAbs}(X,\_), \mathtt{DelPre}(X)\}, \\ & \quad x \in X \text{ and } R = \{(\_, v)\} \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.40}$$

$$\mathsf{wspec}(w)(x,v) = \begin{cases} v' & \text{if } w = \mathtt{InsAbs}(X,v') \wedge v = \dagger \\ \dagger & \text{if } w = \mathtt{DelPre}(X) \wedge v \neq \dagger \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.41}$$

where $\dagger$ is a special value representing absence. We assume that $\mathtt{InsAbs}(X, \dagger)$ is not defined.

The ins/del is maximally layered w.r.t. ar, with 1 as its layer bound. ins/del allows execution-correctors: let CMod be a consistency model, ξ be an abstract execution, D be a domain, $W \subseteq D$ be a set of objects and x be an object s.t. $x \in W$ if $W \neq \emptyset$.

Let be v the value that event e reads in $\xi \oplus e$. If $v = \dagger$, we select $e = \texttt{InsAbs}(D, \_)$ while otherwise, $e = \texttt{DelPre}(D)$. The mapping a below is an execution-corrector for (e, W, x, ξ):

$$a(y) = \begin{cases} \texttt{InsAbs}(\{y\}, v') & \text{if } y \in W \wedge v_y = \dagger \neq v, \text{ or } y \notin W \wedge v_y = \dagger = v \\ \texttt{DelPre}(\{y\}) & \text{if } y \in W \wedge v_y \neq \dagger = v, \text{ or } y \notin W \wedge v_y \neq \dagger \neq v \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.42}$$

where $v_y = \mathsf{wspec}(e_p)(y, [\xi, \mathsf{CMod}])$ and e_p is the maximal event w.r.t. so on the same replica as e.

5.9.5 Non-Transactional SQL with Last-Writer-Wins Store

The Non-Transactional SQL with Last-Writer-Wins Store (simple-SQL) is an operation specification modelling SQL-like databases [9]. Each object represents a row identifier and the set of values is defined abstractly as Rows. Rows contain a special value denoted $\dagger$, different from $\perp$, indicating that the row is deleted.

This operation specification employs four operations: `INSERT`, `SELECT`, `UPSERT` and `DELETE`. Each operation has a finite set of objects D as domain. $\texttt{INSERT}(\mathsf{R})$ inserts in the database each row r on an object $d \in D$ using the mapping $\mathsf{R} : D \to \mathsf{Rows}$. $\texttt{SELECT}(\mathsf{p})$ selects the rows on the storage satisfying the predicate $\mathsf{p} : D \times \mathsf{Rows} \to \{\mathsf{false}, \mathsf{true}\}$. $\texttt{UPSERT}(\mathsf{p}, \mathsf{U})$ updates the rows that satisfy p using the mapping $\mathsf{U} : D \times \mathsf{Rows} \to \mathsf{Rows}$, inserting them if they are absent. Finally, $\texttt{DELETE}(\mathsf{p})$, deletes the objects satisfying the predicate (i.e. replaces its row by $\dagger$). We assume that in for any predicate p and object x, $\mathsf{p}(x, \dagger) = \mathsf{false}$.

$$\mathsf{rspec}(r)(x, c) = \begin{cases} \{\max_{\mathsf{ar}} E\} & \text{if } r \in \{\texttt{SELECT}(\mathsf{p}), \texttt{UPSERT}(\mathsf{p}, \mathsf{U}), \texttt{DELETE}(\mathsf{p})\} \\ & \text{ and } c = (E, \mathsf{ar}, \mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{5.43}$$

$$\mathsf{extract}(r)(x, R) = \begin{cases} v & \begin{array}{l}\text{if } r \in \{\texttt{SELECT}(\mathsf{p}), \texttt{UPSERT}(\mathsf{p}, \mathsf{U}), \texttt{DELETE}(\mathsf{p})\}, \\ \quad R = \{(w, v)\} \text{ and } \mathsf{p}_x(v)\end{array} \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.44}$$

$$\mathsf{wspec}(w)(x, v) = \begin{cases} \mathsf{R}(x) & \text{if } w = \texttt{INSERT}(\mathsf{R}) \\ \mathsf{U}_x(v) & \text{if } w = \texttt{UPSERT}(\mathsf{p}, \mathsf{U}) \\ \dagger & \text{if } w = \texttt{DELETE}(\mathsf{p}) \wedge v \notin \{\perp, \dagger\} \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.45}$$

The simple-SQL is maximally layered w.r.t. ar, with 1 as its layer bound. simple-SQL allows execution-correctors: let CMod be a consistency model, ξ be an abstract execution, D be a domain, $W \subseteq D$ be a set of objects and x be an object s.t. $x \in W$ if $W \neq \emptyset$.

Let be v the value that event e reads in $\xi \oplus e$. We select the event $e = \texttt{UPSERT}(\mathsf{p}_{D,W}, \mathsf{U}_D)$, where $\mathsf{p}_{D,W}$ and U_D are defined below.

$$\mathsf{p}_{D,W}(d,r) = \begin{cases} \mathsf{true} & \text{if } d \in W \\ \mathsf{false} & \text{if } d \in D \setminus W \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$
$$\mathsf{U}_D(d,r) = \begin{cases} r & \text{if } d \in D \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

For such event, we define the execution-corrector $a : D \setminus \{x\} \to \mathsf{Events}$ as the totally-undefined mapping, i.e. the function that no object $y \in D$ is associated with some event.

5.9.6 Transactional SQL Multi-Value Store

The Transactional SQL Multi-Value Store (SQL-mv) is an operation specification modelling SQL-like databases using *transactions*. Each object represents a row identifier and the set of values, Rows, is defined as in Section 5.9.5.

Transactions are blocks of simple instructions that are executed sequentially. Transactions start its execution by selecting a *snapshot* of the database (i.e. a mapping associating each object a constant value) from which operations can read. Each instruction may execute a writing operation, but its effect it is only viewed internally. After their completion, the writing effects of the transaction can be seen by other transactions; giving the impression of atomicity.

We model the store with the aid of a unique operation, `TRANSACTION`(body) that reads the snapshot of the database and then executes the instructions declared in `C`. `C` is defined as a sequence of five type of operations: `INSERT`, `SELECT`, `UPDATE` and `DELETE`. Each operation has a finite set of objects D as domain. `INSERT`(R) inserts in the database each row r on an object $d \in D$ using the mapping $\mathsf{R} : D \to \mathsf{Rows}$. `SELECT`(p) selects the rows on the storage satisfying the predicate $\mathsf{p} : D \times \mathsf{Rows} \to \{\mathsf{false}, \mathsf{true}\}$. `UPDATE`(p, U) updates the rows that satisfy p using the mapping $\mathsf{U} : D \times \mathsf{Rows} \to \mathsf{Rows}$. Finally, `DELETE`(p), deletes the objects satisfying the predicate (i.e. replaces its row by $\dagger$). `abort` represents states declared by the user where the transaction should not execute any more instructions and any declared write should be aborted. We assume that in for any predicate p and object x, $\mathsf{p}(x, \dagger) = \mathsf{false}$.

We model snapshots as mappings $\mathsf{Keys} \to \mathsf{Vals}$. Unlike in Section 5.9.5, SQL-mv requires that local effects of SQL-like instructions are only seen internally, during the execution of the transaction. Such effects are modelled in Equation (5.46) as a recursive function that simulates the transaction execution w.r.t. a concrete object. The function exe executes one instruction at a time, and it stops whenever all instructions are executed, indicating that the execution was correct, or halting it midway in case some abortion occurred (modelled with the constant value $\bot$).

$$\mathsf{exe}_x(\mathsf{body}, \sigma) = \begin{cases} \mathsf{exe}_x(\mathsf{body}', \sigma') & \text{if } \mathsf{body} = e; \mathsf{body}', \sigma' = \mathsf{exI}_x(e, \sigma) \text{ and } \sigma' \neq (\bot, \mathsf{false}) \\ \sigma & \text{if } \mathsf{body} = \emptyset \\ (\bot, \mathsf{false}) & \text{otherwise} \end{cases} \tag{5.46}$$

The behavior of each instruction is modelled in Equation (5.47), updating the snapshot in object x in a similar way as wspec does in Section 5.9.5, and indicating if the event e indeed wrote object x.

$$\mathsf{exI}_x(e,(\sigma,\mathsf{w})) = \begin{cases} (\sigma,\mathsf{w}) & \text{if } e = \mathtt{SELECT}(\mathsf{p}) \\ (\sigma,\mathsf{w}) & \text{if } e = \mathtt{DELETE}(\mathsf{p}) \wedge \neg \mathsf{p}_x(\sigma) \\ (\dagger,\mathsf{true}) & \text{if } e = \mathtt{DELETE}(\mathsf{p}) \wedge \mathsf{p}_x(\sigma) \\ (\sigma,\mathsf{w}) & \text{if } e = \mathtt{UPDATE}(\mathsf{p},\ \mathsf{U}) \text{ and either } \neg \mathsf{p}_x(\sigma) \text{ or } U_x(\sigma)\uparrow \\ (\mathsf{U}_x(\sigma),\mathsf{true}) & \text{if } e = \mathtt{UPDATE}(\mathsf{p},\ \mathsf{U}), \mathsf{p}_x(\sigma) \wedge U_x(\sigma)\uparrow \\ (\sigma,\mathsf{w}) & \text{if } e = \mathtt{INSERT}(\mathsf{R}) \wedge \mathsf{R}(x)\uparrow \\ (\mathsf{R}(x),\mathsf{true}) & \text{if } e = \mathtt{INSERT}(\mathsf{R}) \wedge \mathsf{R}(x)\downarrow \\ (\bot,\mathsf{false}) & \text{if } e = \mathtt{abort} \end{cases} \tag{5.47}$$

The operation specifications of $\mathsf{SQL\text{-}mv}$ are an adaptation of those of $\mathsf{k\text{-}mv}$:

$$\mathsf{rspec}(r)(x,c) = \begin{cases} \{\max_{\mathsf{rb}} E\} & \text{if } r = \mathtt{TRANSACTION}(\mathsf{body}) \text{ and } c = (E,\mathsf{ar},\mathsf{rb}) \\ \emptyset & \text{otherwise} \end{cases} \tag{5.48}$$

$$\mathsf{extract}(r)(x,R) = \begin{cases} \sigma' & \text{if } r = \mathtt{TRANSACTION}(\mathsf{body}), \sigma = \{(v,\mathsf{false}) \mid (w,v) \in R\} \\ & \quad \text{and } \sigma' = \mathsf{exe}_x(\mathsf{body},\sigma) \end{cases} \tag{5.49}$$

$$\mathsf{wspec}(w)(x,\sigma) = \begin{cases} v & \text{if } r = \mathtt{TRANSACTION}(\mathsf{body}),\ \text{and } \sigma = (v,\mathsf{true}) \\ \mathsf{undefined} & \text{otherwise} \end{cases} \tag{5.50}$$

The $\mathsf{SQL\text{-}mv}$ operation specification is maximally layered w.r.t. rb^+, with 1 as its layer bound. $\mathsf{SQL\text{-}mv}$ allows execution-correctors: let CMod be a consistency model, ξ be an abstract execution, D be a domain, $W \subseteq D$ be a set of objects and x be an object s.t. $x \in W$ if $W \neq \emptyset$.

We define $e = \mathtt{TRANSACTION}(\mathtt{SELECT}(\mathsf{p}_D); \mathtt{INSERT}(\mathsf{R}_W))$, where p_W and U_D are defined below.

$$\mathsf{p}_D(d,r) = \begin{cases} \mathsf{true} & \text{if } d \in D \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$
$$\mathsf{R}_W(d) = \begin{cases} \_ & \text{if } d \in D \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

where _ indicates some arbitrary unspecified value.

For such event, we define the execution-corrector $a : D \setminus \{x\} \to \mathsf{Events}$ as the totally-undefined mapping, i.e. the function that no object $y \in D$ is associated with some event.

5.10 The Arbitration-Free Consistency Theorem

We now present our main result in its most general form, which extends Theorem 5.6.3.

Theorem 5.10.1 (Arbitration-Free Consistency (AFC)). *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification. The following statements are equivalent:*

1. CMod *is arbitration-free w.r.t.* OpSpec*,*
2. *there exists an available* OpSpec*-implementation.*

The proof of (1) ⇒ (2) is very similar to that in Theorem 5.6.3 (see Section 5.6.1). The only difference is replacing Lemma 5.6.5 with Lemma 5.8.11 where we use the maximally-layered assumption of read specifications. For the reverse, we follow the reasoning explained in the beginning of Section 5.6.2 to reduce to consistency models in normal form. Lemma 5.10.2 extends the arguments in Lemma 5.6.7 to generalized storage specifications.

Lemma 5.10.2. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification. Assume that* CMod *contains a simple visibility formula* v *which is non-vacuous w.r.t.* OpSpec*, such that for some* $i, 0 \leq i \leq \mathsf{len}(\mathsf{v})$*,* $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$*. Then, there is no available* $(\mathsf{CMod}, \mathsf{OpSpec})$*-implementation.*

Proof Sketch. As in Lemma 5.6.7, we assume by contradiction that there is an available implementation I_E of Spec. We use the visibility formula v to construct a specific program, which by the assumption, admits a trace (in the composition with this implementation) that contains no `receive` action. We show that any abstract execution induced by this trace, which is admissible by any available implementation of Spec, is not valid w.r.t. Spec. This contradicts the hypothesis.

Let d_{v} be the largest index i s.t. $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$ (last occurrence of ar). Then, v is formed of two parts: the path of dependencies from ε_0 to $\varepsilon_{d_{\mathsf{v}}}$ which is not arbitration-free, and the suffix from $\varepsilon_{d_{\mathsf{v}}}$ up to $\varepsilon_{\mathsf{len}(\mathsf{v})}$, the arbitration-free part.

The program P that we construct uses two replicas r_0, r_1, two objects x_0, x_1 and a collection of events $e_i^{x_l}, 0 \leq i \leq \mathsf{len}(\mathsf{v}), l \in \{0, 1\}$. The events are used to "encode" two instances of v_{x_0} and v_{x_1}. Replica r_l executes first events $e_i^{x_l}$ with $i < d_{\mathsf{v}}$ and then, events $e_i^{x_{1-l}}$ with $i \geq d_{\mathsf{v}}$ – the replica r_l executes the non arbitration-free part of v for object x_l and the arbitration-free suffix of v for x_{1-l}. For every l, the event $e_{\mathsf{len}(\mathsf{v})}^{x_l}$ reads x_{1-l}.

For ensuring that $\mathsf{v}_x(e_0^{x_l}, \ldots e_n^{x_l})$ holds in an induced abstract execution of a trace without `receive` actions, we require that if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, then $e_{i-1}^{x_l}$ is a write event and $e_i^{x_l}$ is a read event. For ensuring that $\mathsf{wrCons}_x^{\mathsf{v}}(e_0, \ldots e_{\mathsf{len}(\mathsf{v})})$ holds in such an abstract execution, for each set $E \in \mathcal{P}(\varepsilon_0, \ldots e_{\mathsf{len}(\mathsf{v})})$ s.t. $\mathsf{conflict}(E)$ occurs in v, we consider a distinct object y_E, which is also distinct from x_0 and x_1. These objects represent each conflict in v in a distinct manner. Then, we require that events $e_i^{x_l}$ write to object y_E iff $\varepsilon_i \in E$ and to object x_l iff ε_i belongs to the set E_x s.t. $\mathsf{conflict}_x(E_x)$ occurs in v (since v is conflict-maximal, there is only one occurrence of a $\mathsf{conflict}_x$ predicate). In the case $e_i^{x_l}$ is a conditional read-write, we add a set of events $A_i^{x_l}$ that form an execution-corrector so $\mathsf{conflict}_x(e_0^{x_l}, \ldots e_{\mathsf{len}(\mathsf{v})}^{x_l})$ holds in an abstract execution of a trace without `receive` actions. These additional events do not write on objects x_0 or x_1.

Figure 5.9 exhibits a diagram of the abstract execution of the program.

The rest of the proof, which proceeds as follows, is a generalization of the proof of Lemma 5.6.7 which takes into considerations the assumptions we make about storage specifications:

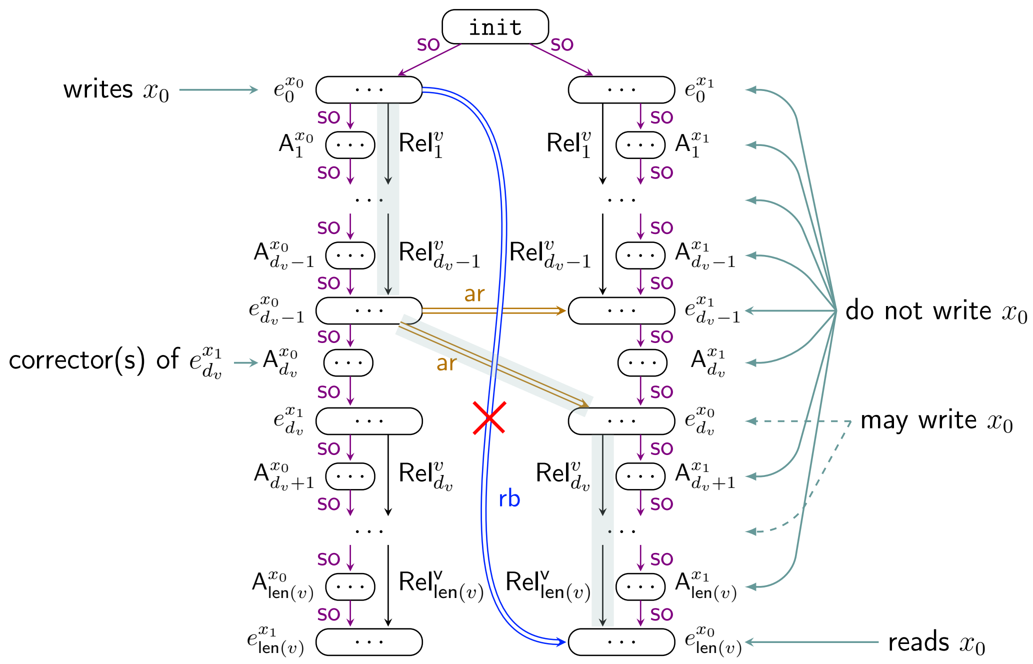


Figure 5.9: Abstract execution of a trace without `receive` actions for the visibility formula v. $\mathsf{A}_i^{x_l}$ represents a sequence of events $a_i^{x_l}(y), y \in \mathsf{obj}(e_i^{x_l})$ associated to an execution-corrector. The auxiliary events in $\mathsf{A}_i^{x_l}$ allow that, if $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}-1}^{x_1}) \in \mathsf{ar}$, $\mathsf{wrCons}_x^\mathsf{v}(e_0^{x_0}, \dots e_{\mathsf{len(v)}}^{x_0})$ holds, and thus $v_{x_0}(e_0^{x_0}, e_{\mathsf{len(v)}}^{x_0})$ holds as well.

1. There exists a finite trace t of $P \parallel I_E$ that contains no receive action (Lemma 5.7.5).

2. The trace t induces a history $h_\mathsf{v} = (E, \mathsf{so}, \mathsf{wr})$ and an abstract execution $\xi_\mathsf{v} = (h, \mathsf{rb}, \mathsf{ar})$ where $\mathsf{rb} = \mathsf{so}$. As I_E is valid w.r.t. Spec, ξ_v is valid w.r.t. Spec. Next, we prove that since $\mathsf{rb} = \mathsf{so}$, events in ξ_v read the latests value w.r.t. so for any object. In particular, we deduce that ξ_v is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ (Corollary 5.12.5).

3. Since ar is a total order, either $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}-1}^{x_1}) \in \mathsf{ar}$ or $(e_{d_\mathsf{v}-1}^{x_1}, e_{d_\mathsf{v}-1}^{x_0}) \in \mathsf{ar}$. W.l.o.g., assume that $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}-1}^{x_1}) \in \mathsf{ar}$. By Proposition 5.12.6, we deduce that $e_0^{x_0} \in \mathsf{ctxt}_{x_0}(e_{\mathsf{len(v)}}^{x_0}, [\xi_\mathsf{v}, \mathsf{CMod}])$. The proof is explained in Figure 5.9: if $(e_{d_\mathsf{v}-1}^{x_0}, e_{d_\mathsf{v}-1}^{x_1}) \in \mathsf{ar}$, then all events $e_i^{x_0}$ form a path in such way that $\mathsf{v}_{x_0}(e_0^{x_0}, \dots e_{\mathsf{len(v)}}^{x_0})$ holds in ξ_v. If some event $e_i^{x_l}$ is a conditional read-write event, the predicate $\mathsf{conflict}_x(e_0^{x_0}, \dots e_{\mathsf{len(v)}}^{x_0})$ holds in ξ_v thanks to the corrector events $A_i^{x_l}$.

4. As $e_0^{x_0} \in \mathsf{ctxt}_{x_0}(e_{\mathsf{len}(v)}^{x_0}, [\xi_\mathsf{v}, \mathsf{CMod}])$ but $(e_0^{x_0}, e_{\mathsf{len(v)}}^{x_0}) \notin \mathsf{rb}$ (no message is received), we deduce in Proposition 5.11.16that OpSpec is layered w.r.t. ar. By contrapositive, if OpSpec would be layered w.r.t. rb, as $e_0^{x_0} \in \mathsf{ctxt}_{x_0}(e_{\mathsf{len(v)}}^{x_0}, [\xi_\mathsf{v}, \mathsf{CMod}])$, there would exist an event e s.t. $(e_0^{x_0}, e) \in \mathsf{rb}$ and $e \in \mathsf{rspec}(e_{\mathsf{len(v)}}^{x_0})(x_0, [\xi_\mathsf{v}, \mathsf{CMod}])$. However, as $\mathsf{rb} = \mathsf{so}$, $\mathtt{rep}(e_0^{x_0}) = \mathtt{rep}(e) = \mathtt{rep}(e_{\mathsf{len(v)}}^{x_0})$ which is false because $\mathtt{rep}(e_0^{x_0}) = r_0$ and

$\mathtt{rep}(e^{x_0}_{\mathsf{len(v)}}) = r_1$.

5. Since rspec is maximally layered, we can show that the layer bound of rspec is smaller than or equal to the number of arbitration-free suffixes of v (Proposition 5.11.17). Observe that an event writes x_0 only if it is $\mathtt{init}$ or is an event $e^{x_l}_i$ s.t. $\varepsilon_i \in E_x$ and $l = 0$. Any such index i corresponds to a suffix of v. By causal suffix closure, for any arbitration-free suffix v' of v there is a visibility formula that subsumes v' in $\mathsf{nCMod}_{\mathsf{OpSpec}}$. As d_{v} is the maximum index for which $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{ar}$, the number of events writing x_0 in replica r_1 distinct from $\mathtt{init}$ coincide with the number of arbitration-free suffixes of v. Hence, as rspec is layered w.r.t. ar, if its layer bound would be greater than the number of arbitration-free suffixes, $e^{x_0}_{\mathsf{len(v)}}$ would necessarily read x_0 from $\mathtt{init}$ (other events writing x_0 are in replica r_0 and $e_{\mathsf{len(v)}}$ only reads from events in r_1). However, as rspec is maximally-layered and $e^{x_0}_0$ succeeds $\mathtt{init}$ w.r.t. ar and rb^+, we would conclude that $e^{x_0}_{\mathsf{len(v)}}$ would also read x_0 from $e^{x_0}_0$. However, this is impossible as $\mathsf{wr} \subseteq \mathsf{rb} = \mathsf{so}$ but $e^{x_0}_0$ is in replica r_0 and $e^{x_0}_{\mathsf{len(v)}}$ is in replica r_1.

6. Lastly, we show in Proposition 5.11.18 that if the layer bound of rspec is smaller than or equal to the number of arbitration-free suffixes of v, then v is vacuous w.r.t. OpSpec, which contradicts the fact that v is a visibility formula from the normal form $\mathsf{nCMod}_{\mathsf{OpSpec}}$. □

Corollary 5.10.3 is an immediate consequence of Theorem 5.10.1 and Lemma 5.6.4.

Corollary 5.10.3. *Let* OpSpec *be an operation specification. The strongest consistency model* CMod *for which* $(\mathsf{CMod}, \mathsf{OpSpec})$ *admits an available implementation is* CC.

5.11 Normal Form of a Consistency Model w.r.t. an Operation Specification

In this section, we prove the existence of a consistency model in normal form equivalent to a given one (Theorem 5.11.1), and we show as well that arbitration-freeness is well-defined (Theorem 5.11.9), i.e. that either all its normal forms are arbitration-free or none.

For compare consistency models when restricted to an operation specification OpSpec, we introduce the notion of OpSpec-equivalence. Two consistency models CMod_1, CMod_2 are OpSpec-*equivalent*, denoted $\mathsf{CMod}_1 \equiv_{\mathsf{OpSpec}} \mathsf{CMod}_2$, if for every abstract execution of OpSpec, ξ, ξ is valid w.r.t. $(\mathsf{CMod}_1, \mathsf{OpSpec})$ iff ξ is valid w.r.t. $(\mathsf{CMod}_2, \mathsf{OpSpec})$. In particular, if CMod_1 and CMod_2 are equivalent, they are also OpSpec-equivalent. The converse is not true: vacuous visibility formulas under an operation specification OpSpec may not be vacuous for every possible operation specification.

5.11.1 Existence of a Normal Form of a Consistency Model

Theorem 5.11.1 states the existence of a normal form of a consistency model w.r.t. OpSpec.

Theorem 5.11.1. *Let* OpSpec *be an operation specification. For every consistency model* CMod*, there exists a consistency model that is in normal form w.r.t.* OpSpec *and that is* OpSpec*-equivalent to* CMod.

The proof of such result is divided in three parts, proving the existence of a consistency model with only simple visibility formulas (Lemma 5.11.4), proving that such model can be refined for removing vacuous visibility formulas (Lemma 5.11.7) and finally, showing that conflict-maximality can be assumed without loss of generality (Lemma 5.11.8).

Monotonicity

Maximally-layered operation specifications are *monotonic*. Intuitively, an operation specification is *monotonic* if (1) the values that are not read under a consistency model CMod_1 should be also not read under a stronger model CMod_2, and (2) whenever some values are read under a consistency model CMod_1 but not under a stronger one CMod_2, some other values must be read under CMod_2 which were not visible under CMod_1.

Definition 5.11.2. *Let* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *be an operation specification.* OpSpec *is called* monotonic *if for every pair of consistency models* $\mathsf{CMod}_1, \mathsf{CMod}_2$*,* $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$*, abstract execution* ξ*, event* $r \in \xi$*, and object* x *the following hold:*

1. $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \subseteq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \cup (\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]))$.
2. *if* $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \neq \emptyset$*, then* $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \neq \emptyset$

Lemma 5.11.3. *A maximally-layered operation specification is monotonic.*

Proof. Let OpSpec be a maximally-layered operation specification, $\mathsf{CMod}_1, \mathsf{CMod}_2$ be two consistency models s.t. $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$, ξ be an abstract execution, r be an event in ξ and x be an object. Observe that by the unconditional read property of OpSpec (Property 2 of Definition 5.8.4), we can assume w.l.o.g. that r is a read event.

On one hand, we observe that if the layer bound of OpSpec is ∞, OpSpec is trivially monotonic: as r is a read event and the layer bound of OpSpec is ∞, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$ and $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. Using the fact that $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$, is easy to see that Properties 1 and 2 hold in this case.

On the other hand, if the layer bound of OpSpec, k, is finite, let R be the relation for which OpSpec is k-maximally layered. For proving Property 1 of Definition 5.11.2, let us partition $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$ in the three disjoint sets C_1, C_2 and C_3 described in Equation (5.51).

$$\begin{aligned} C_1 &\coloneqq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \\ C_2 &\coloneqq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \\ C_3 &\coloneqq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \end{aligned} \tag{5.51}$$

We note that by Property 1 of Definition 5.8.4, we know that $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. As $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$, we deduce that $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$; so $\{C_1, C_2, C_3\}$ is indeed a partition of $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$. Observe that showing Property 1 of Definition 5.11.2 is equivalent to show that $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \subseteq C_1 \cup C_3$. By Property 1 of Definition 5.8.4, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2]) = C_1 \cup C_2 \cup C_3$. We conclude the result by showing that $C_2 \cap \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) = \emptyset$.

For showing it, we observe that the layer of an event w in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$ is less or equal than the layer of w in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$: as $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$, every chain of events in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$ containing w and ordered w.r.t. R belongs to $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$. Thus, as OpSpec is maximally layered, an event w in C_2 does not belong to $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$: if $w \in C_2$, its layer in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$ is greater than k; so it is also greater than k in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$. Hence, as OpSpec has k as layer bound, $w \notin \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$.

For proving Property 2, we observe that if there exists an event w in the set $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$, then the layer of w in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_2])$ is greater than k. Let k' be the layer of w and let $\{e_i\}_{i=1}^{k'}$ be a chain of R of length k' s.t. $e_{k'} = w$. As the layer of w in $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$ is k and R is a partial order, there exists an event $e_i, 1 \leq i \leq k$ s.t. $e_i \in C_3$. We observe that as the layer of w is k, the layer of event e_i is i. Hence, as rspec is k-maximally layered, we conclude that $e_i \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. □

Lemma 5.8.11 shows that for maximally-layered operation specifications, ensuring a strong consistency criteria is enough for ensuring a weaker one. The proof relies on the fact that maximally-layered operation specifications are monotonic (Lemma 5.11.3).

Lemma 5.8.11. *Let* OpSpec *be a maximall-layered operation specification and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be a pair of consistency models such that* CMod_2 *is stronger than* CMod_1*. Any abstract execution valid w.r.t.* $(\mathsf{CMod}_2, \mathsf{OpSpec})$ *is also valid w.r.t.* $(\mathsf{CMod}_1, \mathsf{OpSpec})$.

Proof. Let $h = (E, \mathsf{so}, \mathsf{wr})$ be a history and let CMod_1 and CMod_2 be two consistency models s.t. $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$. Let also $\xi = (h, \mathsf{rb}, \mathsf{ar})$ be an abstract execution that witness the validity of h w.r.t. $(\mathsf{CMod}_2, \mathsf{OpSpec})$. To prove that ξ also witnesses h's validity w.r.t. $(\mathsf{CMod}_1, \mathsf{OpSpec})$, by Definition 5.8.8, it suffices to prove that for every event $r \in h$ and object x, $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])$.

- $\underline{\mathsf{wr}_x^{-1}(r) \subseteq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])}$: Let w be a write event in $\mathsf{wr}_x^{-1}(r)$. As $(w, r) \in \mathsf{wr}_x$, $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. Moreover, as ξ witnesses h's validity w.r.t. CMod_2, $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$. Hence, as $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \cap \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$, by Property 1 of Definition 5.11.2, $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])$.

- $\underline{\mathsf{wr}_x^{-1}(r) \supseteq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])}$: Let $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])$ s.t. $w \notin \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$. By property 2 from Definition 5.11.2, there exists $w' \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2])$ s.t. $w' \notin \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$. However, as $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) = \mathsf{wr}_x^{-1}(r) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$, this is impossible. Therefore, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \subseteq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) = \mathsf{wr}_x^{-1}(r)$. □

An immediate consequence of Lemma 5.8.11 is the following result.

Lemma 5.6.5. *Let* OpSpec *be a basic operation specification, and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be a pair of basic consistency models s.t.* CMod_2 *is weaker than* CMod_1*. Any abstract execution valid w.r.t.* $(\mathsf{CMod}_2, \mathsf{OpSpec})$ *is also valid w.r.t.* $(\mathsf{CMod}_1, \mathsf{OpSpec})$.

Simple Form

For proving Theorem 5.11.1, we first prove the existence of a consistency model in simple form (i.e. a consistency model with all its visibility formulas are simple) that is equivalent to CMod.

Lemma 5.11.4. *For any consistency model* CMod*, there exists a consistency model in simple form that is equivalent to* CMod*.*

Intuitively, the proof of Lemma 5.11.4 is as follows: we first unfold union and transitive closure operators, and then trim `id` and compositional operators to obtain a consistency model in simple form. As an intermediate step, we define the consistency model obtained after unfolding union and transitive closure operators. Such consistency model is the *almost simple form* of CMod, $\mathsf{almost}(\mathsf{CMod})$, and it is described as the union of the *almost simple form* of each of its visibility formulas, i.e. $\mathsf{almost}(\mathsf{CMod}) = \bigcup_{v \in \mathsf{CMod}} \mathsf{almost}(v)$. A visibility formula a belongs to the almost simple form of a visibility formula v, $a \in \mathsf{almost}(v)$ if (1) $\mathsf{len}(v) = \mathsf{len}(a)$ and (2) for every $i, 1 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^a \in \sigma(\mathsf{Rel}_i^{\mathsf{v}})$; where $\sigma(\mathsf{Rel}i^v)$ is the set of relations described as follows:

$$\sigma(\mathsf{R}) = \begin{cases} \{\mathsf{R}\} & \text{if } \mathsf{R} = \mathtt{id}, \mathsf{so}, \mathsf{wr}, \mathsf{rb} \text{ or } \mathsf{ar} \\ \sigma(\mathsf{S}) \cup \sigma(\mathsf{T}) & \text{if } \mathsf{R} = \mathsf{S} \cup \mathsf{T} \\ \sigma(\mathsf{S}); \sigma(\mathsf{T}) & \text{if } \mathsf{R} = \mathsf{S}; \mathsf{T} \\ \bigcup_{k \in \mathbb{N} \wedge k \geq 1} \sigma(\mathsf{S})^k & \text{if } \mathsf{R} = \mathsf{S}^+ \end{cases} \tag{5.52}$$

where the composition of two sets of relations A, B is defined as $A; B ::= \{a; b \mid a \in A, b \in B\}$.

We prove that CMod and $\mathsf{almost}(\mathsf{CMod})$ are equivalent.

Proposition 5.11.5. *For any consistency model* CMod*,* CMod *and* $\mathsf{almost}(\mathsf{CMod})$ *are equivalent.*

Proof. For proving the result, we show that for any abstract execution ξ, object x and event r, $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{almost}(\mathsf{CMod})])$. In particular, it suffices to prove that for every visibility formula $v \in \mathsf{CMod}$ and event w, $v_x(w, r)$ holds in ξ iff there exists a visibility formula $a \in \mathsf{almost}(v)$ s.t. $a_x(w, r)$ holds in ξ. Observe that for every $a \in \mathsf{almost}(v)$, $\mathsf{len}(v) = \mathsf{len}(a)$; so we reduce the proof to show that for every pair of events e, e', $(e, e') \in \mathsf{Rel}_i^{\mathsf{v}}$ iff there exists $\mathsf{R}' \in \sigma(\mathsf{Rel}_i^{\mathsf{v}})$ s.t. $(e, e') \in \mathsf{R}'$.

In the following, we prove that for every relation R over pair of events obtained by the grammar described in Equation (5.3), the following holds: $(e, e') \in \mathsf{R}$ iff there exists $\mathsf{R}' \in \sigma(R)$ s.t. $(e, e') \in \mathsf{R}'$. We show the result by induction on the depth of R[6]. The base case, when the depth of R is 0, refers to the case $\mathsf{R} = \mathtt{id}, \mathsf{so}, \mathsf{wr}, \mathsf{rb}, \mathsf{ar}$. In such case, the result immediately holds by the definition of $\sigma(\mathsf{R})$.

Let us assume that for any relation of depth at most n the result holds, and let us prove that for relations of depth $n + 1$. Three alternatives arise:

- If $\mathsf{R} = \mathsf{S} \cup \mathsf{T}$, $(e, e') \in \mathsf{R}$ if and only if $(e, e') \in \mathsf{S} \cup \mathsf{T}$. By induction hypothesis on both S and T, $(e, e') \in \mathsf{S} \cup \mathsf{T}$ iff there exists $\mathsf{R}' \in \sigma(\mathsf{S}) \cup \sigma(\mathsf{T})$ s.t. $(e, e') \in \mathsf{R}'$. Finally, by

[6]By depth of R we mean the depth of the tree obtained by deriving R using Equation (5.3).

Equation (5.52), we conclude that there exists $\mathsf{R}' \in \sigma(\mathsf{S}) \cup \sigma(\mathsf{T})$ s.t. $(e, e') \in \mathsf{R}'$ if and only if there exists $\mathsf{R}' \in \sigma(\mathsf{R})$ s.t. $(e, e') \in \mathsf{R}'$.

- If $\mathsf{R} = \mathsf{S}; \mathsf{T}$, $(e, e') \in \mathsf{R}$ if and only if $(e, e') \in \mathsf{S}; \mathsf{T}$. By the definition of composition, $(e, e') \in \mathsf{S}; \mathsf{T}$ iff there exists e'' s.t. $(e, e'') \in \mathsf{S}$ and $(e'', e') \in \mathsf{T}$. By induction hypothesis on both S and T, there exists e'' s.t. $(e, e'') \in \mathsf{S}$ and $(e'', e') \in \mathsf{T}$ iff there exists e'' and relations $\mathsf{S}' \in \sigma(\mathsf{S}), \mathsf{T}' \in \sigma(\mathsf{T})$ e'' s.t. $(e, e'') \in \mathsf{S}'$ and $(e'', e') \in \mathsf{T}'$. By the definition of $\sigma(S); \sigma(T)$, we observe that there exists e'' and relations $\mathsf{S}' \in \sigma(\mathsf{S}), \mathsf{T}' \in \sigma(\mathsf{T})$ e'' s.t. $(e, e'') \in \mathsf{S}'$ and $(e'', e') \in \mathsf{T}'$ iff there exists relation $\mathsf{R}' \in \sigma(\mathsf{S}; \mathsf{T})$ s.t. $(e, e') \in \mathsf{R}'$. Finally, by Equation (5.52), we conclude that there exists relation $\mathsf{R}' \in \sigma(\mathsf{S}; \mathsf{T})$ s.t. $(e, e') \in \mathsf{R}'$ if and only if there exists $\mathsf{R}' \in \sigma(\mathsf{R})$ s.t. $(e, e') \in \mathsf{R}'$.

- If $\mathsf{R} = \mathsf{S}^+$, $(e, e') \in \mathsf{R}$ if and only if there exists $k \in \mathbb{N}^+$ s.t. $(e, e') \in \mathsf{S}^k$. By the previous point, there exists $k \in \mathbb{N}^+$ s.t. $(e, e') \in \mathsf{S}^k$ if and only if there exists $k \in \mathbb{N}^+$ and relation $\mathsf{S}' \in \sigma(\mathsf{S})^k$ s.t. $(e, e') \in \mathsf{S}'$. Finally, by Equation (5.52), we conclude that there exists $k \in \mathbb{N}^+$ and relation $\mathsf{S}' \in \sigma(\mathsf{S})^k$ s.t. $(e, e') \in \sigma(\mathsf{S})^k$ if and only if there exists relation $\mathsf{R}' \in \sigma(\mathsf{R})$ s.t. $(e, e') \in \mathsf{R}'$.

□

Obtaining a consistency model in simple form from a consistency model in almost simple form is straightforward: every visibility formula is transformed by splitting composed relations into simpler subrelations and omitting `id` by merging two existentially quantified events. Lemma 5.11.4 formally describes such procedure.

Lemma 5.11.4. *For any consistency model* CMod*, there exists a consistency model in simple form that is equivalent to* CMod*.*

Proof. We construct a consistency model, $\mathsf{simple}(\mathsf{CMod})$, that is in simple form and it is equivalent to CMod. The model is formally defined as follows:

$$\mathsf{simple}(\mathsf{CMod}) = \{\mathsf{simple}(a) \mid a \in \mathsf{almost}(\mathsf{CMod})\} \tag{5.53}$$

where $\mathsf{simple}(a)$ is the *simple visibility formula* of a.

The simple visibility formula of a visibility formula in almost form a is the visibility formula f obtained by supressing `id` and compositional operators. Formally, f is the visibility formula s.t. (1) $\mathsf{len}(f) = \sum_{i=1}^{\mathsf{len}(a)} \mathsf{count}(\mathsf{Rel}_i^a)$ and (2) for every $i, 1 \leq i \leq \mathsf{len}(f)$, $\mathsf{Rel}_i^f = \mathsf{rel}(\mathsf{Rel}_j^a, i - k_j)$; where j is the maximum index s.t. $k_j < i$ and $k_j = \sum_{l=1}^{j} \mathsf{count}(\mathsf{Rel}_l^a)$, and count and rel are the functions described in Equation (5.54) and Equation (5.55) respectively.

The function count counts the number of additional quantifiers the correspondant simple form requires:

$$\mathsf{count}(\mathsf{R}) = \begin{cases} 0 & \text{if } \mathsf{R} = \mathtt{id} \\ 1 & \text{if } \mathsf{R} = \mathsf{so}, \mathsf{wr}, \mathsf{rb} \text{ or } \mathsf{ar} \\ \mathsf{count}(\mathsf{S}) + \mathsf{count}(\mathsf{T}) & \text{if } \mathsf{R} = \mathsf{S}; \mathsf{T} \end{cases} \tag{5.54}$$

Also, the function rel, given a relation using compositional operator and an index i, returns the i-th component:

$$\mathsf{rel}(\mathsf{R}, i) = \begin{cases} \mathsf{R} & \text{if } \mathsf{R} = \mathsf{so}, \mathsf{wr}, \mathsf{rb} \text{ or } \mathsf{ar} \\ \mathsf{rel}(\mathsf{S}, i) & \text{if } i \leq \mathsf{count}(\mathsf{S}) \\ \mathsf{rel}(\mathsf{T}, i - \mathsf{count}(\mathsf{S})) & \text{otherwise} \end{cases} \tag{5.55}$$

By construction, $\mathsf{simple}(\mathsf{CMod})$ is in simple form. Clearly, $\mathsf{simple}(\mathsf{CMod})$ is equivalent to $\mathsf{almost}(\mathsf{CMod})$. Then, thanks to Proposition 5.11.5, we conclude that $\mathsf{simple}(\mathsf{CMod})$ is equivalent to CMod. □

Removing Vacuous Visibility Formulas

After proving the existence of a consistency model CMod in simple form equivalent to a given one, we show how to transform it for obtaining an equivalent consistency model CMod without vacuous visibility formulas (Lemma 5.11.7). We say that any such consistency model is in *basic normal form*, extending Definition 5.6.1 to any consistency model whose visibility formulas are described using Equation (5.15).

The following result, key to prove Lemma 5.11.7, it is a simple consequence of Definition 5.11.2 and Lemma 5.11.3.

Proposition 5.11.6. *Let* OpSpec *be a maximally-layered operation specification, and let* $\mathsf{CMod}_1, \mathsf{CMod}_2$ *be two consistency models s.t.* $\mathsf{CMod}_1 \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}_2$ *but* $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$*. There exists an abstract execution* ξ *valid w.r.t.* CMod_1*, an object* x *and events* w, r *s.t.* $w \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1])$.

Proof. First of all, as $\mathsf{CMod}_1 \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}_2$ but $\mathsf{CMod}_1 \preccurlyeq \mathsf{CMod}_2$, by Lemma 5.8.11, there exists an abstract execution ξ valid w.r.t. CMod_1, an object x and an event r s.t. $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \neq \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1])$. Thus, either $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \neq \emptyset$ or $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \neq \emptyset$.

On one hand, if $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \neq \emptyset$, by Property 1 of Definition 5.11.2, then $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \neq \emptyset$. On the other hand, if $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_1]) \setminus \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \neq \emptyset$, by Property 2 of Definition 5.11.2, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}_2]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_1]) \neq \emptyset$. □

Lemma 5.11.7. *Let* OpSpec *be an operation specification. For every consistency model* CMod *in simple form, there exists a* OpSpec*-equivalent consistency model,* $\mathsf{bnCMod}_{\mathsf{OpSpec}}$*, that is in basic normal form w.r.t.* OpSpec.

Proof. To prove the result, we construct a consistency model in basic normal form w.r.t. OpSpec, $\mathsf{bnCMod}_{\mathsf{OpSpec}}$, that is OpSpec-equivalent to CMod. Without loss of generality we can assume that CMod is ordered. Let α be an ordinal of cardinality $|\mathsf{CMod}|$. We denote by $v^i, 0 \leq i < \alpha$ to the i-th visibility formula in CMod[7].

We construct a sequence of nested consistency models $\mathsf{CMod}_k, 0 \leq k \leq \alpha$ s.t. (1) CMod_k is OpSpec-equivalent to CMod, (2) CMod_k is more succinct than CMod_i (i.e., for every $i < k$,

[7]Without loss of generality, we can assume that limit ordinals in α are not associated to a visibility formula
.

$v^i \in \mathsf{CMod}_k$ iff $v^i \in \mathsf{CMod}_i$ and for every $i > k$, $v^i \in \mathsf{CMod}_k$), and (3) the first k visibility formulas of CMod_k are simple and non-vacuous w.r.t. $(\mathsf{CMod}_k, \mathsf{OpSpec})$ (i.e., for every $i, 0 \leq i < k$, if $v^i \in \mathsf{CMod}_k$, then $\mathsf{CMod}_k \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$).

We construct such sequence using transfinite induction. The base case, $k = 0$, corresponds to $\mathsf{CMod}_0 = \mathsf{CMod}$, which trivially satisfies (1), (2) and (3). For the successor case, let us assume that the property holds for the consistency model CMod_k, and let us prove it for CMod_{k+1}. If $\mathsf{CMod}_k \setminus \{v^k\} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$, we denote CMod_{k+1} as $\mathsf{CMod}_k \setminus \{v^k\}$; and otherwise, $\mathsf{CMod}_{k+1} = \mathsf{CMod}_k$.

Clearly, by construction of CMod_{k+1}, (1) and (2) immediately hold. For proving (3), we observe that if $v^i \in \mathsf{CMod}_{k+1}$, $v^i \in \mathsf{CMod}_i$. In such case, $\mathsf{CMod}_i \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$. Hence, by Lemma 5.8.11, there exists an abstract execution valid w.r.t. $(\mathsf{CMod}_i \setminus \{v^i\}, \mathsf{OpSpec})$ that is not valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$. As $\mathsf{CMod}_{k+1} \subseteq \mathsf{CMod}_i$, $\mathsf{CMod}_{k+1} \setminus \{v^i\} \subseteq \mathsf{CMod}_i \setminus \{v^i\}$ and hence, $\mathsf{CMod}_{k+1} \setminus \{v^i\} \preccurlyeq \mathsf{CMod}_i \setminus \{v^i\}$. Therefore, by Lemma 5.8.11, ξ is valid w.r.t. $(\mathsf{CMod}_{k+1} \setminus \{v^i\}, \mathsf{OpSpec})$. Thus, as ξ is not valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$, $\mathsf{CMod}_{k+1} \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$; so we conclude (3).

For the limit case, we define CMod_k as the intersection of all consistency models $\mathsf{CMod}_i, i < k$. We observe that in this case, (2) immediately holds by construction of CMod_k.

For proving (3) we observe that $v^i \in \mathsf{CMod}_k$ iff $v^i \in \mathsf{CMod}_i$. In such case, $\mathsf{CMod}_i \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$; so by Lemma 5.8.11, there exists an abstract execution ξ valid w.r.t. $(\mathsf{CMod}_i \setminus \{v^i\}, \mathsf{OpSpec})$ that is not valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$. Similarly to the inductive case, we deduce using Lemma 5.8.11 that ξ is valid w.r.t. $(\mathsf{CMod}_k \setminus \{v^i\}, \mathsf{OpSpec})$. Therefore, we conclude that $\mathsf{CMod}_i \setminus \{v^i\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$.

For proving (1), we reason by contradiction, assuming that $\mathsf{CMod}_k \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$ and reaching a contradiction. In such case, by Lemma 5.8.11 there exists an abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ valid w.r.t. $(\mathsf{CMod}_k, \mathsf{OpSpec})$ that is not valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$. W.l.o.g., we can assume that ξ is minimal w.r.t. the number of events in it; and let $\mathsf{len}(\xi)$ the number of events in such execution.

For each event $r \in \xi$, we define an ordinal $i(r), i(r) < k$ associated to every visibility formula $v^i, i < k$ that can be applied on ξ. First, we note that for every pair of events, e, e' and object x, if a visibility formula $v_x(e, e')$ holds in ξ, $\mathsf{len}(v) \leq \mathsf{len}(\xi)$. Observe that there exists finite number of visibility formulas v in CMod with at most length $\mathsf{len}(\xi)$: on one hand, for each $j, 1 \leq j \leq \mathsf{len}(v)$, Rel_j^v is either $\mathsf{so}, \mathsf{wr}, \mathsf{rb}$ or ar. On the other hand, wrCons is defined as a conjunction of predicates from a finite set. Thus, the number of possible visibility formulas v of length $\mathsf{len}(v) \leq \mathsf{len}(\xi)$ is finite. Let i_r be the biggest index of a visibility formula $v^i \in \mathsf{CMod}$ s.t. $\mathsf{len}(v^i) \leq \mathsf{len}(\xi)$ and $i < k$; and let $i(r) = i_r + 1$. Observe that k is a limit ordinal, $i(r) < k$.

Let x be an object and r be an event in ξ. We show that $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}])$. As $\mathsf{CMod}_k \subseteq \mathsf{CMod}_{i(r)}$, $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}])$. For showing $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}]) \subseteq \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k])$, let $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}])$. In such case, there exists a visibility formula v^i s.t. $v^i(w, r)$ holds in ξ. If $i > k$, by (2) $v^i \in \mathsf{CMod}_k$. Otherwise, $i < i(r)$, so by (2), $v^i \in \mathsf{CMod}_i$. Observe that in this case, applying the induction hypothesis (2) on every consistency model $\mathsf{CMod}_j, j < k$, $v^i \in \mathsf{CMod}_j$, we deduce that $v^i \in \mathsf{CMod}_k$. Either way, we deduce that $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k])$. In conclusion,

$\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_k]) = \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}_{i(r)}])$.

We conclude a contradiction by showing that ξ is valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$; which by assumption it is not. Let e be the last event w.r.t. ar in ξ. For reaching such contradiction, as $i(e) < k$ and $\mathsf{CMod}_{i(e)} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$, it suffices to show that ξ is valid w.r.t. $(\mathsf{CMod}_{i(e)}, \mathsf{OpSpec})$. We show that $\mathsf{wr}_x^{-1}(e') = \mathsf{rspec}(e')(x, [\xi, \mathsf{CMod}_{i(e)}])$.

On one hand, if $e' = e$, we note that $\mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}_k]) = \mathsf{ctxt}_x(e, [\xi, \mathsf{CMod}_{i(e)}])$. As ξ is valid w.r.t. $(\mathsf{CMod}_k, \mathsf{OpSpec})$, we conclude that $\mathsf{rspec}(e)(x, [\xi, \mathsf{CMod}_{i(e)}]) = \mathsf{wr}_x^{-1}(e)$.

On the other hand, if $e' \neq e$, let ξ' be the execution obtained by removing e from ξ. By the minimality of ξ, ξ' is valid w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$. By induction hypothesis (1), $\mathsf{CMod} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}_{i(e)}$. Hence, ξ' is valid w.r.t. $(\mathsf{CMod}_{i(e)}, \mathsf{OpSpec})$. We thus deduce that $\mathsf{wr}_x^{-1}(e') = \mathsf{rspec}(e')(x, [\xi, \mathsf{CMod}_{i(e)}])$. In conclusion, CMod_k satisfies (1) and thus, the inductive step.

Finally, we define $\mathsf{bnCMod}_{\mathsf{OpSpec}} = \mathsf{CMod}_\alpha$. As CMod_α satisfies (1) and (3), it is a consistency model OpSpec-equivalent to CMod composed of finite, non-vacuous w.r.t. $(\mathsf{CMod}_\alpha, \mathsf{OpSpec})$ visibility formulas; so we conclude that it is a consistency model in basic normal form.

□

Conflict-Strengthening a Consistency Model

Lemma 5.11.8. *Let* OpSpec *be an operation specification. For every consistency model* CMod *in basic normal form w.r.t.* OpSpec *there exists a* OpSpec*-equivalent consistency model that is in normal form.*

Proof. We transform CMod to define $\mathsf{nCMod}_{\mathsf{OpSpec}}$, a consistency model in normal form that is OpSpec-equivalent to CMod.

For every visibility formula $v \in \mathsf{CMod}$, we define v' as the visibility formula that only differs with v on its conflict predicate. More specifically, we require that for every set $E \in \mathcal{P}(\varepsilon_0, \ldots \varepsilon_{\mathsf{len}(v)})$, we require that $\mathsf{conflict}(E) \in v'$ (resp. $\mathsf{conflict}_x(E) \in v'$) iff (1) for every abstract execution ξ, every object x and every collection of events $e_0, \ldots e_{\mathsf{len}(v)}$ s.t. $v_x(e_0, \ldots e_{\mathsf{len}(v)})$ holds in ξ_v, there exists an object $y \neq x$ s.t. if $\varepsilon_i \in E, 0 \leq i \leq \mathsf{len}(v)$, then $\mathsf{wspec}(e_i)(y, [\xi, \mathsf{CMod}]) \downarrow$ (resp. $\mathsf{wspec}(e_i)(x, [\xi, \mathsf{CMod}]) \downarrow$) and (2) there is no strict superset of E satisfying (1). We define $\mathsf{nCMod}_{\mathsf{OpSpec}}$ as the set containing all such visibility formulas. For conclude the result, we first prove that $\mathsf{nCMod}_{\mathsf{OpSpec}} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$ for then deduce that $\mathsf{nCMod}_{\mathsf{OpSpec}}$ is indeed a consistency model in normal form.

We show that $\mathsf{nCMod}_{\mathsf{OpSpec}} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$. On one hand, as every visibility formula v' enforces more conflicts than v, $\mathsf{nCMod}_{\mathsf{OpSpec}} \preccurlyeq \mathsf{CMod}$. On the other hand, by the definition of v', for every abstract execution ξ, object x and events w, r, if $v'_x(w, r)$ holds in ξ, $v_x(w, r)$ also holds in ξ. Altogether, we conclude that $\mathsf{nCMod}_{\mathsf{OpSpec}} \equiv_{\mathsf{OpSpec}} \mathsf{CMod}$.

To show that $\mathsf{nCMod}_{\mathsf{OpSpec}}$ is a consistency model in normal form, we observe that by construction, every visibility formula $v \in \mathsf{nCMod}_{\mathsf{OpSpec}}$ is in simple form and it is conflict-maximal w.r.t. OpSpec. Hence, it suffices to prove that every visibility formula $v \in \mathsf{nCMod}_{\mathsf{OpSpec}}$ is non-vacuous w.r.t. $\mathsf{nCMod}_{\mathsf{OpSpec}}$.

Let v' be a visibility formula of $\mathsf{nCMod}_{\mathsf{OpSpec}}$. Observe that by construction of $\mathsf{nCMod}_{\mathsf{OpSpec}}$, $\mathsf{nCMod}_{\mathsf{OpSpec}} \setminus \{v'\} \equiv \mathsf{CMod} \setminus \{v\}$. Hence, as $\mathsf{CMod} \setminus \{v\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$, we deduce that $\mathsf{nCMod}_{\mathsf{OpSpec}} \setminus \{v'\} \equiv \mathsf{CMod} \setminus \{v\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod} \equiv \mathsf{nCMod}_{\mathsf{OpSpec}}$. In other words, v' is non-vacuous w.r.t. $(\mathsf{nCMod}_{\mathsf{OpSpec}}, \mathsf{OpSpec})$.

$\square$

5.11.2 Arbitration-Free Well-Formedness

As described in Section 5.8.1, a consistency model is arbitration-free if a OpSpec-equivalent consistency model in normal form is arbitration-free. In Theorem 5.11.9, we present a result that states that arbitration-free is well-defined, as either every OpSpec-equivalent consistency model in normal form are arbitration-free or none.

Regarding notations, for a visibility formula v and $i, 0 \leq i \leq \mathsf{len}(v)$ we denote hereinafter $\mathsf{conflictsOf(v,i)} \in \mathcal{P}(\mathcal{P}(\varepsilon_0, \ldots \varepsilon_{\mathsf{len}(v)}))$ to the sets of conflicts of ε_i in v, i.e. $E \in \mathsf{conflictsOf(v,i)}$ iff $\varepsilon_i \in E$ and $\mathsf{conflict}(E) \in v$.

Theorem 5.11.9. *Let* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *be an operation specification and let* CMod *be a consistency model. For every pair of consistency models in normal form* n_1, n_2 *that are* OpSpec*-equivalent to* CMod*,* n_1 *is arbitration-free iff* n_2 *is arbitration-free.*

Proof. We prove the result by contradiction, assuming that there exists two consistency models n_1, n_2 in normal form, OpSpec-equivalent to CMod, but one of them arbitration-free and the other one no. W.l.o.g., we can assume that n_1 is arbitration-free and n_2 is not. On one hand, as n_2 is not arbitration-free w.r.t. OpSpec, there exists a visibility formula $v \in n_2$ s.t. v is not arbitration-free. We construct an abstract execution that is valid w.r.t. (n_1, OpSpec) but not valid w.r.t. (n_2, OpSpec) using v, reaching a contradiction.

First of all, observe that by Lemma 5.6.4, n_1 is weaker than $\mathtt{CC}$. The abstract execution we construct contains a collection events $e_0, \ldots e_{\mathsf{len}(v)}$s.t. ξ is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ and $v_x(e_0, \ldots e_{\mathsf{len}(v)})$ holds on it; for some object x.

Let x be an object. For each set $E \in \mathcal{P}(\varepsilon_0, \ldots e_{\mathsf{len}(v)})$ we consider a distinct object y_E, also distinct from x. These objects represents each different conflict in v in an explicit manner.

We denote by $E_x \in \mathcal{P}(\varepsilon_0, \ldots e_{\mathsf{len}(v)})$ to the set s.t. $\mathsf{conflict}_x(E_x) \in v$. Also, for every $i, 0 \leq i \leq \mathsf{len}(v)$, we denote by X_i to the set containing objects y_E (resp. x) iff $E \in \mathsf{conflictsOf(v,i)}$ (resp. $E_x \in \mathsf{conflictsOf(v,i)}$). We denote by X to the union of sets $X_i, 0 \leq i \leq \mathsf{len}(v)$.

For obtaining ξ, we construct a sequence of executions $\xi^i, 0 \leq i \leq \mathsf{len}(v)$ inductively, starting from an initial event $\mathtt{init}$, and incorporating at each time a new event e_i. We use the notation h^{-1} and ξ^{-1} to describe the history and abstract execution containing only $\mathtt{init}$ respectively. We use the convention $e_{-1} = \mathtt{init}$, $\mathsf{conflictsOf(v,-1)} = \mathsf{Keys}$ and $\tilde{x}_{-1} = o_{-1} = x$ (the usage of such conventions will be clearer later).

For the inductive step, we assume that the abstract execution $\xi^{i-1} = (h^{i-1}, \mathsf{rb}^{i-1}, \mathsf{ar}^{-1})$ associated to the history $h^{i-1} = (E^{i-1}, \mathsf{so}^{i-1}, \mathsf{wr}^{i-1})$ contains events $e_{-1} \ldots e_{i-1}$ and is well-defined (satisfies Definition 5.3.4) and we construct the history h^i and the abstract execution ξ^i. First of all, we impose the constraint that if $i > 0$, then $r_i = r_{i-1}$ iff $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{so}$, and otherwise $r_i \neq r_j, 0 \leq j < i$.

Also, we define a pair of special objects, $\tilde{x}_i$ and o_i. The purpose of object $\tilde{x}_i$ is control the number of events in ξ that write object x. Equation (5.56) describes $\tilde{x}_i$; where choice is a function that deterministically chooses an element from a non-empty set. The object o_i is an object different from objects $x, y_E, E \in \mathcal{P}(\varepsilon_0, \dots \varepsilon_{\mathsf{len}(v)})$ and $o_j, -1 \leq j < i$ that we use for ensuring that if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, then $(e_{i-1}, e_i) \in \mathsf{wr}$.

$$\tilde{x}_i = \begin{cases} \tilde{x}_{i-1} & \text{if } X_i = \emptyset \\ x & \text{if } X_i \neq \emptyset \text{ and } x \in X_i \\ \mathsf{choice}\,(X_i) & \text{if } X_i \neq \emptyset \text{ and } x \notin X_i \end{cases} \tag{5.56}$$

We select a domain D_i, a set of objects $W_i, W_i \subseteq D_i$ that event e_i must write, and a set of objects $C_i \subseteq D_i$ whose value needs to be corrected for e_i in ξ_{i+1} – in the sense of Definition 5.8.13. We distinguishing between several cases:

- $\underline{i = 0 \text{ or } 0 < i \leq \mathsf{len}(v) \text{ and } \mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr} \text{ and } \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset}$: In this case, we select e_i to be a write event. If OpSpec only allows single-object atomic read-write events, we define $D_i = X_i$; while if not, we consider a domain containing o_{i-1}, o_i, every object in X_i but no object from $X \setminus X_i$ nor objects $o_j, 0 \leq j < \mathsf{len}(v), j \neq i-1, i$. Observe that by Proposition 5.11.10, such domain always exist on OpSpec.

  If there is an unconditional write event whose domain is D_i, we define $W_i = D_i$. Otherwise, we define $W_i = X_i \cup \{o_i\}$.

- $\underline{0 < i \leq \mathsf{len}(v), \mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr} \text{ and } \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset}$: In this case, by Proposition 5.11.11, OpSpec allows atomic read-write events. If OpSpec only allows single-object atomic read-write events, we define $D_i = X_i$; while if not, we consider a domain containing o_{i-1}, o_i, every object in X_i but no object from $X \setminus X_i$ nor objects $o_j, 0 \leq j < \mathsf{len}(v), j \neq i-1, i$. Observe that by Proposition 5.11.10, such domain always exist on OpSpec.

  Similarly to the previous case, if there is an unconditional atomic read-write event whose domain is D_i, we define $W_i = D_i$. Otherwise, we define $W_i = X_i \cup \{o_i\}$.

- $\underline{0 < i \leq \mathsf{len}(v) \text{ and } \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) = \emptyset}$: In this case, by Proposition 5.11.11, OpSpec allows events that do not unconditionally write. If OpSpec allows read events that are not write events, we select D_i to be the domain of any such event and $W_i = \emptyset$. Otherwise, OpSpec must allow conditional write events; so we select D_i to be the domain of any such event, $W_i = \emptyset$. Observe that in this case, thanks to the assumptions on OpSpec (see Section 5.8.4), we can assume without loss of generality that whenever $o_{i-1} \in D_{i-1}$, $o_{i-1} \in D_i$ as well; while otherwise, that $\tilde{x}_{i-1} \in D_i$.

Finally we describe the event e_i thanks to the sets D_i and W_i. If $W_i = D_i$ and $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, we select an unconditional atomic read-write event whose domain is D_i. If $W_i = D_i$ and $\mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr}$, we select an unconditional write event whose domain is D_i. If $W_i = \emptyset$ and OpSpec allows read events that are not write events, we select a read event whose domain is D_i. Finally, if that is not the case, we select a conditional write event e_i s.t. $\mathsf{obj}(e_i) = D_i$ and s.t. an execution-corrector exists for $(e_i, W_i, \tilde{x}_i, \xi^{i-1} \oplus e_i)$. Such event always exists by the assumptions on operation specifications (Section 5.8.4). W.l.o.g. we can assume that e_i happens on replica r_i.

For concluding the description of $h^i = (E_i, \mathsf{so}^i, \mathsf{wr}^i)$ and $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$, we use an auxiliary history and abstract execution, $h^i_0 = (E^i_0, \mathsf{so}^i_0, \mathsf{wr}^i_0)$ and $\xi^i_0 = (h^i_0, \mathsf{rb}^i_0, \mathsf{ar}^i_0)$ respectively. For describing the write-read dependencies of e_i in ξ^0_i, we define the context mapping $c^i : \mathsf{Keys} \to \mathsf{Contexts}$, associating each object y to the context $c^i(y)$ described in Equation (5.57).

$$c^i(y) = (F^i(y), \mathsf{rb}^{i-1}_{\restriction F^i(y)\times F^i(y)}, \mathsf{ar}^{i-1}_{\restriction F^i(y)\times F^i(y)}) \tag{5.57}$$

where $F^i(y)$ is the mapping associating each object y with the set of events described below:

$$F^i(y) = \begin{cases} \{\mathtt{init}\} & \text{if } i = 0 \\ & \text{ or if } 0 < i \leq \mathsf{len}(v) \land \mathsf{Rel}_i = \mathsf{ar} \\ \left\{ e \in E^{i-1} \;\middle|\; \begin{array}{l} \mathsf{wspec}(e)(y, [\xi^{i-1}, \mathtt{CC}]) \downarrow \text{and} \\ (e, e_{i-1}) \in (\mathsf{rb}^{i-1})^* \end{array} \right\} & \text{otherwise} \end{cases}$$

Then, we define ξ^i_0 as the abstract execution of the history $h^i_0 = (E^i_0, \mathsf{so}^i_0, \mathsf{wr}^i_0)$ obtained by appending e_i to h^i_0 and ξ^i_0 as follows: E^i_0 contains E^{i-1} and event e_i. First of all, we require that the relations so^i_0, wr^i_0, rb^i_0 and ar^i_0 contain so^{i-1}, wr^{i-1}, rb^{i-1} and ar^{i-1} respectively. With respect to event e_i, we impose that e_i is the maximal event w.r.t. so^i_0 among those on the same replica. Also, e_i is maximal w.r.t. wr as we define that for every object z, ${\mathsf{wr}^i_{0z}}^{-1}(e_i) = \mathsf{rspec}(e_i)(z, c_i(z))$. For describing rb^i_0, we require that for every event e s.t. $(e, e_i) \in \mathsf{so}^i_0$, $(e, e_i) \in \mathsf{rb}^i$. Also, if $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{rb}$, we impose that $(e_{i-1}, e_i) \in \mathsf{rb}^i_0$. Finally, we require that for every pair of events $e, e' \in E^{i-1}$ s.t. $(e, e') \in \mathsf{rb}^{i-1}$ and $(e', e_i) \in \mathsf{so}^i_0$, $(e, e_i) \in \mathsf{rb}^i_0$. With respect to ar^i_0, we impose that e_i is the maximum event w.r.t. ar in ξ^i_0.

We use ξ^i_0 to construct ξ^i. If event e_i is not a conditional write event, $\xi^i = \xi^i_0$. Otherwise, if event e_i is a conditional write event, given W_i and object $\tilde{x}_i$, we select an execution-corrector for e_i w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ and a_i. W.l.o.g., we assume that every event mapped by a_i happens on replica r_i. Observe that by the choice of sets D_i and W_i, and thanks to the assumptions on storages (see Section 5.8.4), such event(s) are always well-defined.

In addition, we denote by C_i to the set of objects we need to correct for e_i. More specifically, if e_i is a conditional write-read, we denote by C_i to the set of objects y s.t. $a_i(y)$ is defined, i.e. $C_i = \{y \in \mathsf{Keys} \mid a_i(y) \downarrow\}$. In the case e_i is not a conditional write-read, we use the convention $C_i = \emptyset$. The set of events in ξ^i is the following: $E^i = E^{i-1} \cup \{e_i\} \bigcup_{y \in C_i \setminus \{o_{i-1}\}} a_i(y)$. Observe that by the choice of C_i, the set E^i is well-defined.

Concerning notations, we use $c \oplus a$ to denote the context obtained by appending a to the context $c = \{E, \mathsf{rb}, \mathsf{ar}\}$ as the rb-maximum and ar-maximum event.

From ξ^i_0, we define $\xi^i = \xi^i_0 \overset{\mathsf{seq}(a_i)}{\curlyvee} e_i$ as the corrected execution of ξ and e_i with events a_i. For describing ξ^i, we consider $<$ to be a well-founded order over Keys. ξ^i satisfies the following:

- $\underline{\mathsf{so}^i}$: Let $y \in C_i$. We require that for every event $e \in E^{i-1}$, $(e, a_i(y)) \in \mathsf{so}^i$ iff $\mathtt{rep}(e) = r_i, 0 \leq j < i$. We also require that $(\mathtt{init}, a_i(y)) \in \mathsf{so}^i$ and $(a_i(y), e_i) \in \mathsf{so}^i$. Finally, we require that for every objects $y' \in C_i, y' < y$, $(a_i(y'), a_i(y)) \in \mathsf{so}^i$.

- wr^i: Let y be an object in C_i. For every object z, if $z \in C_i$ and $z < y$, we require that $(\mathsf{wr}^i_z)^{-1}(a_i(y)) = \mathsf{rspec}(a_i(y))(z, c^i(z) \oplus a_i(z))$; while otherwise, we require that $(\mathsf{wr}^i_z)^{-1}(a_i(y)) = \mathsf{rspec}(a_i(y))(z, c^i(z))$. We also require that for every object z, if $z \in C_i$, then $(\mathsf{wr}^i_z)^{-1}(e_i) = \mathsf{rspec}(e_i)(z, c^i(z) \oplus a_i(z))$, while otherwise, $(\mathsf{wr}^i_z)^{-1}(e_i) = \mathsf{rspec}(e_i)(z, c^i(z))$.

- rb^i: Let $y \in C_i$. We require that for every object $y \in C_i$ and event e s.t. $(e, a_i(y)) \in \mathsf{so}^i \cup \mathsf{wr}^i$, $(e, a_i(y)) \in \mathsf{rb}^i$. Also, if $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{rb}$, we impose that $(e_{i-1}, a_i(y)) \in \mathsf{rb}^i$. Finally, we require that for every pair of events $e, e' \in E^{i-1}$ s.t. $(e, e') \in \mathsf{rb}^{i-1}$ and $(e', a_i(y)) \in \mathsf{so}^i$, $(e, a_i(y)) \in \mathsf{rb}^i$.

- ar^i: We impose that for every event $e \in E^{i-1}$, $(e, a_i(y)) \in \mathsf{ar}^i, y \in C_i$. We also require that for every pair of objects $y_1, y_2 \in C_i$ s.t. y_1, y_2, $(a_i(y_1), a_i(y_2)) \in \mathsf{ar}^i$.

We then define $h^i = (E^i, \mathsf{so}^i, \mathsf{wr}^i)$ and $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$. Observe that by construction of h^i and ξ^i, they satisfy Definitions 5.3.2 and 5.3.4 respectively; so they are a history and an abstract execution respectively. In particular, observe that ξ^i is a correction of the abstract execution ξ^{i-1}_0 with events a_i.

Finally, we define $h = (E, \mathsf{so}, \mathsf{wr})$ and $\xi = (h, \mathsf{rb}, \mathsf{ar})$ as, respectively, the history $h^{\mathsf{len}(v)}$ and the abstract execution $\xi^{\mathsf{len}(v)}$. We prove that ξ is the abstract execution we were looking for.

First, we show that ξ is valid w.r.t. n_2: as ξ is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ (Corollary 5.11.13), so by Lemma 5.6.4, it is valid w.r.t. (n_1, OpSpec). As $n_1 \equiv_{\mathsf{OpSpec}} n_2$, ξ is valid w.r.t. (n_2, OpSpec). Next, we deduce in Proposition 5.11.16 that OpSpec is maximally layered w.r.t. ar. For proving such result, we rely on Propositions 5.11.14 and 5.11.15. Finally, we conclude in Proposition 5.11.17 that the layer bound of rspec is bounded by the number of arbitration-free suffixes of v. However, this implies that v is vacuous w.r.t. n_2 (Proposition 5.11.18); which is impossible by the choice of v. The contradiction arises from assuming that n_1 is arbitration-free but n_2 is not; so we conclude the result. $\square$

Proposition 5.11.10. *Let* OpSpec *be a storage that allows multi-object write (resp. read-write) events whose domain is not* Keys*. Then, for every pair of finite disjoint sets* F_1, F_2 *there exists a domain* D *in* OpSpec *s.t.* $F_1 \subseteq D$ *but* $F_2 \cap D = \emptyset$.

Proof. The result is immediate as F_1 is finite. Hence, by the assumptions on operation specifications (Section 5.8.4), F_1 is a domain on OpSpec. $\square$

Proposition 5.11.11. *Let* v *be a visibility formula and* $i, 0 < i \leq \mathsf{len}(v)$*. If* $\mathsf{conflictsOf(v,i)} \neq \emptyset$ *and* $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$*,* OpSpec *allows read-write events. If* $\mathsf{conflictsOf(v,i)} = \emptyset$ *allows events that do not unconditionally write.*

Proof. Observe that as v is non-vacuous w.r.t. $(\mathsf{CMod}, \mathsf{OpSpec})$, $\mathsf{CMod} \setminus \{v\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$. By Proposition 5.11.6, there exists an execution $\overline{\xi}$ valid w.r.t. $\mathsf{CMod} \setminus \{v\}$, an object z and events $f_0, \ldots f_{\mathsf{len}(v)}$ s.t. $v_z(f_0, \ldots f_{\mathsf{len}(v)})$ holds in $\overline{\xi}$.

On one hand, if $\mathsf{conflictsOf(v,i)} \neq \emptyset$ and $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, as $\overline{\xi}$ is valid w.r.t. $\mathsf{CMod} \setminus \{v\}$, there exists z s.t. $\mathsf{rspec}(f_i)(z, [\overline{\xi}, \mathsf{CMod} \setminus \{v\}]) \neq \emptyset$. Also, $\mathsf{conflictsOf(v,i)} \neq \emptyset$ iff f_i writes on some object z'. Hence, f_i is a read-write event.

On the other hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) = \emptyset$, as v is conflict-maximal w.r.t. OpSpec, event f_i does not necessarily write any object. Thus, OpSpec allows events that do not unconditionally write. □

Proposition 5.11.12. *The abstract execution ξ described in Theorem 5.11.9 satisfies that for every $i, 0 \leq i \leq \mathsf{len}(v)$:*

1. *For every object $y \in C_i$, the following conditions hold:*
   (a) *For every object $z \in \mathsf{Keys}$, if $z \in C_i$ and $z < y$, $G(a_i(y), z) = F^i(z) \cup \{a_i(z)\}$, while otherwise, $G(a_i(y), z) = F^i(z)$.*
   (b) *The execution $\xi^i \upharpoonright y$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$.*
2. *For the event e_i, the following conditions hold:*
   (a) *For every object z, if $z \in C_i$, $G(e_i, z) = F^i(z) \cup \{a_i(z)\}$, while otherwise $G(e_i, z) = F^i(z)$.*
   (b) *The execution ξ^i is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$.*

where $\mathsf{ctxt}_z(e, [\xi, \mathsf{CC}]) = (G(e, z), \mathsf{rb}_{\upharpoonright G(e,z) \times G(e,z)}, \mathsf{ar}_{\upharpoonright G(e,z) \times G(e,z)})$.

Proof. We prove the result by induction. In particular, we show that for every $i, -1 \leq i \leq \mathsf{len}(v)$ and object y, either (0) $i = -1$ or (1) and (2) hold. The base case, $i = -1$, is immediate as (0) holds; so let us suppose that the result holds for every $j, -1 \leq j < i$, and let us prove it for i.

For proving the inductive step, we first prove (1) and then (2). As both (1) and (2) have an identical proof (observe that the role of object y in the former is just to declare that event $a_i(y)$ is well-defined), we present only the proof of (1).

We show (1) by transfinite induction. Let α be an ordinal of cardinality $|\mathsf{Keys}|$. For every $k, 0 \leq k \leq \alpha$, we denote by V_k to the set containing the first k elements in Keys according to $<$. We show that (1) holds for every $y \in V_k \cap C_i$.

The base, V_0 is immediate as $V_0 = \emptyset$. We thus focus on the successor case (i.e., showing that if (1) holds for every object $y \in V_k \cap C_i$ it also holds for V_{k+1}), as the limit case is immediate: if k is a limit ordinal, $V_k = \bigcup_{i, i<k} V_i$; so (1) immediately holds. For showing that (1) holds for every object $y \in V_{k+1} \cap C_i$, as by induction hypothesis it holds for every object $y \in V_k \cap C_i$, it suffices to show it for the only object $y \in V_{k+1} \setminus V_i$. W.l.o.g., we can assume that $y \in C_i$; as otherwise the result is immediate.

We first prove (1a) and then we show (1b). Let $z \in \mathsf{Keys}$ be an object. Two cases arise depending on $\mathsf{Rel}_i^{\mathsf{v}}$.

On one hand, if $i = 0$ or $i > 0 \land \mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$, $F^i(z) = \{\mathtt{init}\}$. As $\mathtt{init} \in G(a_i(y), z)$, it suffices to show that the only non-initial event in E in $G(a_i(y), z)$ is $a_i(z)$ (whenever $z \in C_i$ and $z < y$). Observe that an event e belongs to $G(a_i(y), z)$ if $\mathsf{wspec}(e)(z, [\xi, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in \mathsf{rb}^+$. As $a_i(y) \in E^i$, by construction of ξ, e must belong to E^i, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in (\mathsf{rb}^i)^+$.

Observe that as either $i = 0$ or $0 < i \leq \mathsf{len}(v) \land \mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$, by definition of rb^i, $e \notin E^{i-1}$. Thus, e must be an event in $E^i \setminus E^{i-1}$. Observe that by construction of ξ, as $(e, a_i(y)) \in (\mathsf{rb}^i)^+$, such event must be an event $a_i(w), w \in C_i, w < y$. As $\xi^i = \xi_0^i \overset{a_i}{\curlyvee} e_i$, by induction hypothesis (1b), we deduce that $\xi^i \upharpoonright w$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$. Hence, as $\mathsf{wspec}(a_i(w))(z, [\xi^i, \mathsf{CC}]) \downarrow$, we deduce thanks to Property 1 of Definition 5.8.13 that $z = w$ – so $z \in C_i$ and $z < y$.

On the other hand, if $0 < i \leq \mathsf{len}(v) \wedge \mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{ar}$, two sub-cases arise: $z \in C_i, z < y$ or not. Both cases are identical, so we present the former, i.e., if $z \in C_i, z < y$, then $F^i(z) \cup \{a_i(z)\} = G(a_i(y), z)$.

For proving that $F^i(z) \cup \{a_i(z)\} \subseteq G(a_i(y), z)$, we split the proof in two blocks: showing that $F^i(z) \subseteq G(a_i(y), z)$ and showing that $a_i(z) \in G(a_i(y), z)$.

For showing that $F^i(z) \subseteq G(a_i(y), z)$, let e be an event in $F^i(z)$. In such case, to $e \in E^{i-1}$, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $(e, e_{i-1}) \in (\mathsf{rb}^i)^*$. By the construction of ξ, it is easy to see that any such event belongs to E^i, $\mathsf{wspec}(e)(z, [\xi, \mathsf{CC}]) \downarrow$ and $(e, e_{i-1}) \in \mathsf{rb}^*$. As $\mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{ar}$, we deduce that $(e_{i-1}, a_i(y)) \in \mathsf{rb}^i \subseteq \mathsf{rb}$. Hence, $(e, a_i(y)) \in \mathsf{rb}^+$; so $e \in G(a_i(y), z)$. This show that $F^i(z) \subseteq G(a_i(y), z)$.

For showing that $a_i(z) \in G(a_i(y), z)$, we observe that $\xi^i = \xi_0^i \overset{a_i}{\curlyvee} e_i$. As $z < y$, by induction hypothesis (1b), $\xi^i \upharpoonright z$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$. Thus, by Property 1 of Definition 5.8.13, $\mathsf{wspec}(a_i(z))(z, [\xi^i, \mathsf{CC}]) \downarrow$. Hence, $\mathsf{wspec}(a_i(z))(z, [\xi, \mathsf{CC}]) \downarrow$. As $z < y$, $(a_i(z), a_i(y)) \in \mathsf{so}^i \subseteq \mathsf{so}$; so we conclude that $a_i(z) \in G(a_i(y), z)$.

We conclude the proof of the inductive step of (1a) by showing the converse i.e. $F^i(z) \cup \{a_i(z)\} \supseteq G(a_i(y), z)$. Let $e \in G(a_i(y), z)$. First of all, by the definition of Causal visibility formula (see Figure 5.4b), $e \in G(a_i(y), z)$ iff $\mathsf{wspec}(e)(z, [\xi, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in \mathsf{rb}^+$. Observe that if $(e, a_i(y)) \in \mathsf{rb}^+$, by construction of ξ, such event must belong to E^i, $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$ and $(e, a_i(y)) \in (\mathsf{rb}^i)^+$. We prove that if $e \in E^{i-1}$ then $e \in F^i(z)$, while otherwise, if $e \in E^i \setminus E^{i-1}$, then $e = a_i(z)$.

If $e \in E^{i-1}$, as $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$, $\mathsf{wspec}(e)(z, [\xi^{i-1}, \mathsf{CC}]) \downarrow$. Also, as $\mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{ar}$ and $(e, a_i(y)) \in (\mathsf{rb}^i)^+$, we deduce that $(e, e_{i-1}) \in (\mathsf{rb}^{i-1})^*$. In other words, $e \in F^i(z)$.

Otherwise, if $e \in E^i \setminus E^{i-1}$, we note that by construction of ξ, the only events in $E^i \setminus E^{i-1}$ s.t. $(e, a_i(y)) \in (\mathsf{rb}^i)^+$ are events $a_i(w), w \in C_i, w < y$. As $\xi^i = \xi_0^i \overset{\mathsf{seq}(a_i)}{\curlyvee} e_i$ and $z < y$, $\xi^i \upharpoonright z$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$. Thus, as $\mathsf{wspec}(e)(z, [\xi^i, \mathsf{CC}]) \downarrow$, by Property 1 of Definition 5.8.13 we conclude that $e = a_i(z)$.

For concluding the inductive step, we show that (1b) holds. This is immediate by the definition of wr^i: for every event $e \in \xi^i \upharpoonright y$, by induction hypothesis (1a) or (2a) – depending on whether $e = e_j$ or $a_j(w)$, where $0 \leq j \leq i, w \in C_i$ – $(\mathsf{wr}^i)_z^{-1}(e) = \mathsf{rspec}(e)(\mathsf{CC}, [\xi^i \upharpoonright y, z])$. Thus, $\xi^i \upharpoonright y$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$.

□

A consequence of Proposition 5.11.12 is the following result.

Corollary 5.11.13. *The abstract execution ξ described in Theorem 5.11.9 is valid w.r.t.* $(\mathsf{CC}, \mathsf{OpSpec})$.

Proposition 5.11.14. *The predicate $v_{x_0}(e_0, \ldots e_{\mathsf{len}(v)})$ holds in the abstract execution ξ described in Theorem 5.11.9.*

Proof. The proof is a simple consequence of ξ's construction. To show that $v_{x_0}(e_0, \dots e_{\mathsf{len}(v)})$ holds in ξ, we first show that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e_{i-1}, e_i) \in \mathsf{Rel}^{\mathsf{v}}_i$ and to then prove that $\mathsf{wrCons}^{\mathsf{v}}_x(e_0, \dots e_{\mathsf{len}(v)})$ holds in ξ.

We prove that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e_{i-1}, e_i) \in \mathsf{Rel}^{\mathsf{v}}_i$. Four cases arise depending on $\mathsf{Rel}^{\mathsf{v}}_i$.

- $\underline{\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{so}}$: In this case, by construction of events e_{i-1}, e_i, we know that $r_i = r_{i-1}$. Hence, $(e_{i-1}, e_i) \in \mathsf{so}^i \subseteq \mathsf{so}$.

- $\underline{\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}}$: In this case, we first show that there is an object $y \in D_i \cap W_{i-1} \setminus C_i$, and then show that $(e_{i-1}, e_i) \in \mathsf{wr}_y$. For showing the first part, we distinguish between cases depending on whether $o_{i-1} \in D_i$ or not.

  - $\underline{o_{i-1} \in D_i}$: In this sub-case, we show that $y = o_{i-1}$. On one hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) = \emptyset$, by the choice of event e_i, $o_{i-1} \in D_{i-1} \setminus C_i$. On the other hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset$, as $o_{i-1} \in D_i$, we deduce that OpSpec allows multi-object read-write events. Observe that as v is conflict-maximal w.r.t. OpSpec, $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}-1) \neq \emptyset$. Hence, as OpSpec allows multi-object read-write events, we deduce that $o_{i-1} \in D_{i-1} \setminus C_i$. In both cases, as $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}-1) \neq \emptyset$ and $o_{i-1} \in D_{i-1}$, by the choice of W_{i-1}, we conclude that $o_{i-1} \in W_{i-1}$.
  - $\underline{o_{i-1} \notin D_i}$: In this case, we show that $y = \tilde{x}_i$. On one hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) = \emptyset$, $X_i = \emptyset$; so by the choice of $\tilde{x}_i$ (see Equation (5.56)), $\tilde{x}_i = \tilde{x}_{i-1}$. By the choice of D_i, $\tilde{x}_{i-1} \in D_i \setminus C_i$. Moreover, as v is conflict-maximal w.r.t. OpSpec, $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}-1) \neq \emptyset$; so $\tilde{x}_{i-1} \in X_{i-1}$. By the choice of event e_{i-1}, $X_{i-1} \subseteq W_{i-1}$. Altogether, we conclude that $\tilde{x}_i \in W_{i-1}$.

    On the other hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset$, we note that $\tilde{x}_i \in D_i \setminus C_i$. As $o_{i-1} \notin D_i$, we deduce that OpSpec only allows single-object read-write events. Thus, $D_i = \{\tilde{x}_i\}$. As v is conflict-maximal w.r.t. OpSpec, we deduce that $X_i \subseteq X_{i-1}$. As by the choice of e_{i-1}, $X_{i-1} \subseteq W_{i-1}$, we conclude that $\tilde{x}_i \in W_{i-1}$.

  We prove now that $(e_{i-1}, e_i) \in \mathsf{wr}_y$. First, we show that e_{i-1} writes y in ξ. On one hand, if e_{i-1} is an unconditional write event, $\mathsf{wspec}(e_{i-1})(y, c^i(y)) \downarrow$. On the other hand, if e_{i-1} is a conditional write event, as ξ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$ (Corollary 5.11.13) and $y \in W_i$, by Property 2 of Definition 5.8.13, we deduce that $\mathsf{wspec}(e_{i-1})(y, c^i(y)) \downarrow$. Then, as $\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{wr}$, $e_{i-1} \in F^i(y)$. Observe that by construction of ξ, e_{i-1} is the ar-maximum event in $c^i(y)$. We note that as $y \notin C_i$, by Proposition 5.11.12, $F^i(y) = G(e_i, y)$. To sum up, e_{i-1} is the ar-maximum event in $\mathsf{ctxt}_y(e_i, [\xi, \mathsf{CC}])$. As rspec is maximally layered, we deduce that $e_{i-1} \in \mathsf{rspec}(e_i)(y, [\xi, \mathsf{CC}])$. Finally, as ξ is valid w.r.t. CC (Corollary 5.11.13), we conclude that $(e_{i-1}, e_i) \in \mathsf{wr}_y$.

- $\underline{\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{rb}}$: In this case, we explicitly stated that $(e_{i-1}, e_i) \in \mathsf{rb}^i \subseteq \mathsf{rb}$ during the construction of ξ.

- $\underline{\mathsf{Rel}^{\mathsf{v}}_i = \mathsf{ar}}$: Similarly, by definition of ar^i, we know that $(e_{i-1}, e_i) \in \mathsf{ar}^i \subseteq \mathsf{ar}$.

For showing that show that $\mathsf{wrCons}^{\mathsf{v}}_x(e_0, \dots e_{\mathsf{len}(v)})$, we show that for every $i, 0 \leq i \leq \mathsf{len}(v)$ and every set $E \in \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i})$, the event e_i writes on object y_E[8]. If e_i is an unconditional write, by the choice of e_i, it writes on every object in D_i. As $y_E \in D_i$, we conclude that e_i writes on y_E. Otherwise, if e_i is a conditional write, we observe that $y_E \in W_i$. Hence, as $\xi^i = \xi^i_0 \overset{\mathsf{seq}(a_i)}{\curlyvee} e_i$ and ξ^i is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ (Proposition 5.11.12), we deduce using Property 2 of Definition 5.8.13 that $\mathsf{wspec}(e_i)(y_E, [\xi^i, \mathtt{CC}]) \downarrow$. By construction of ξ, we conclude that $\mathsf{wspec}(e_i)(y_E, [\xi, \mathtt{CC}]) \downarrow$. □

Proposition 5.11.15. *Let ξ be the abstract execution described in Theorem 5.11.9. For every $i, 0 \leq i < \mathsf{len}(v)$, if the ε_i suffix of v is non-arbitration-free, then $(e_i, e_{\mathsf{len}(v)}) \notin \mathsf{rb}^+$.*

Proof. The proof is just an observation about the construction of ξ: for every $j, 0 < j \leq \mathsf{len}(v)$, $(e_{j-1}, e_j) \in \mathsf{rb}$ iff $\mathsf{Rel}^{\mathsf{v}}_i \neq \mathsf{ar}$. Hence, $(e_i, e_{\mathsf{len}(v)} \in \mathsf{rb}^+)$ iff for every $j, i < j \leq \mathsf{len}(v)$, $\mathsf{Rel}_j \neq \mathsf{ar}$. In particular, if the ε_i suffix of v is non-arbitration-free, then $(e_i, e_{\mathsf{len}(v)}) \notin \mathsf{rb}^+$. □

Proposition 5.11.16. *Let OpSpec be a storage, CMod be a consistency model in normal form w.r.t. OpSpec and v be a visibility formula in CMod. If there exists an abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ valid w.r.t. CMod, an object x and events w, r s.t. $v_x(w, r)$ holds in ξ but $(w, r) \notin (\mathsf{rb})^+$, then OpSpec is maximally layered w.r.t. ar.*

Proof. First of all, as $v_x(w, r)$ holds in ξ, $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}])$. If OpSpec would be maximally layered w.r.t. $(\mathsf{rb})^+$, $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}])$ contains at least the first layer of $\mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}])$ w.r.t. rb. Hence, there would exist an event w' s.t. $w' \in \mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}])$ and $(w, w') \in (\mathsf{rb})^+$. As ξ is valid w.r.t. CMod, we deduce that $(w', r) \in \mathsf{wr}$. By Definition 5.3.4, we deduce that $(w', r) \in \mathsf{rb}$. However, this implies that $(w, w') \in \mathsf{rb}^+$; which contradicts the assumptions. Hence, OpSpec must be maximally layered w.r.t. ar. □

Proposition 5.11.17. *Let OpSpec be a storage maximally layered w.r.t. ar, CMod be a consistency model in normal form w.r.t. OpSpec and v be a non-arbitration free visibility formula in CMod. Let us suppose that there exists an abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ valid w.r.t. CMod, an object x and events $e_0, \dots e_{\mathsf{len}(v)}$ satisfying the following:*

1. *for every non-initial event e in ξ, if $e \notin \{e_i \mid 0 \leq i \leq \mathsf{len}(v)\}$, then e does not write on x in ξ,*
2. *$v_x(e_0, \dots e_{\mathsf{len}(v)})$ holds in ξ, and*
3. *for every non-arbitration-free ε_k-suffix of v, $(e_k, e_{\mathsf{len}(v)}) \notin \mathsf{rb}^+$.*

In such case, the layer bound of OpSpec is bounded by the number of arbitration-free suffixes of v.

[8]For simplifying the proof, we abuse of notation and say that $y_E = x$ if $E = E_x$. Observe that v is conflict-maximal w.r.t. OpSpec, either $\mathsf{conflict}_x(E_x)$ or $\mathsf{conflict}(E_x)$ do not belong to v.

Proof. We reason by contradiction, assuming that k is bigger than the number of saturable suffixes of v. We first show that $\mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ contains less than k events in $\{e_i \mid 0 \leq i < \mathsf{len}(v)\}$, for then deduce that $\mathtt{init} \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$. After that, we reach a contradiction by showing that $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ but $(e_0, e_{\mathsf{len}(v)}) \notin \mathsf{wr}$; which contradicts that ξ is valid w.r.t. CMod.

We first show that $\mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ contains less than k events in $\{e_i \mid 0 \leq i < \mathsf{len}(v)\}$. As v contains less than k saturable suffixes, by the Assumption 3, there is less than k events in $\{e_i \mid 0 \leq i < \mathsf{len}(v)\}$ that write on x in ξ and that succeed $e_{\mathsf{len}(v)}$ w.r.t. rb^+. As $\mathsf{wr} \subseteq \mathsf{rb}$ (see Definition 5.3.4), we deduce that $\mathsf{wr}_x^{-1}(e_{\mathsf{len}(v)})$ contains less than k events in $\{e_i \mid 0 \leq i < \mathsf{len}(v)\}$. As ξ is valid w.r.t. CMod, $\mathsf{wr}_x^{-1}(e_{\mathsf{len}(v)}) = \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$; so we prove the first part.

For showing that $\mathtt{init} \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$, we observe that by the Assumption 1, no other non-initial event in ξ writes on x in ξ. Hence, $\mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ contain less than k non-initial events. As $\mathtt{init} \in \mathsf{ctxt}_x(e_{\mathsf{len}(v)}, [\xi, \mathsf{CMod}])$, and OpSpec is maximally layered with layer bound k, we conclude that $\mathtt{init} \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$.

For proving that $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$ but $(e_0, e_{\mathsf{len}(v)}) \notin \mathsf{wr}$, we observe that by the Assumption 2, $e_0 \in \mathsf{ctxt}_x(e_{\mathsf{len}(v)}, [\xi, \mathsf{CMod}])$. As OpSpec is maximally layered w.r.t. ar, $\mathtt{init} \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$, $(\mathtt{init}, e_0) \in \mathsf{ar}$ and $e_0 \in \mathsf{ctxt}_x(e_{\mathsf{len}(v)}, [\xi, \mathsf{CMod}])$; we conclude that $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$.

For reaching a contradiction, we observe that v is non-arbitration-free. Hence, by the Assumption 3, $(e_0, e_{\mathsf{len}(v)}) \notin \mathsf{rb}$. Once again, as $\mathsf{wr} \subseteq \mathsf{rb}$ (see Definition 5.3.4), we deduce that $e_0 \notin \mathsf{wr}_x^{-1}(e_{\mathsf{len}(v)})$. However, as $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(x, [\xi, \mathsf{CMod}])$, we conclude that ξ is not valid w.r.t. CMod; which is contradicts the hypothesis. Thus, the layer bound of OpSpec is bounded by the number of arbitration-free suffixes of v. □

Proposition 5.11.18. *Let* $\mathsf{OpSpec} = (E, \mathsf{rspec}, \mathsf{extract}, \mathsf{wspec})$ *be an operation specification maximally layered w.r.t.* ar*,* CMod *be a consistency model in normal form w.r.t.* OpSpec *and* v *be a simple, conflict-maximal w.r.t.* OpSpec*, non-arbitration-free visibility formula. If the layer bound of* rspec *is smaller or equal by the number of arbitration-free suffixes of* v*, then* $v \notin \mathsf{CMod}$*.*

Proof. Let v be a simple, conflict-maximal w.r.t. OpSpec, non-arbitration-free visibility formula. We show that v is vacuous w.r.t. CMod; so $v \notin \mathsf{CMod}$.

We reason by contradiction, assuming that v is non-vacuous w.r.t. CMod. In such case, $\mathsf{CMod} \setminus \{v\} \not\equiv_{\mathsf{OpSpec}} \mathsf{CMod}$ but $\mathsf{CMod} \setminus \{v\} \preccurlyeq \mathsf{CMod}$. By Proposition 5.11.6, there exists an abstract execution on OpSpec, and object x, and events w, r s.t. $\mathsf{rspec}(r)(x, [\xi, \mathsf{CMod}]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod} \setminus \{v\}])$.

We observe that by Property 2 of Definition 5.8.4, $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}])$. Hence, as $w \in \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod}]) \setminus \mathsf{ctxt}_x(r, [\xi, \mathsf{CMod} \setminus \{v\}])$, we deduce that $v_x(w, r)$ holds in ξ. As v is simple, there exist events $e_0, \ldots e_{\mathsf{len}(v)}$ s.t. $e_0 = w$, $e_{\mathsf{len}(v)} = r$ and $v_x(e_0, \ldots e_{\mathsf{len}(v)})$ holds in ξ.

First of all, as rspec is maximally layered w.r.t. ar and $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(e_0, [\xi, \mathsf{CMod}])$, every event in $\{e_0, \ldots e_{\mathsf{len}(v)}\}$ that writes x is also in $\mathsf{rspec}(e_{\mathsf{len}(v)})(e_0, [\xi, \mathsf{CMod}])$. As v is conflict-maximal w.r.t. OpSpec, at least $|E_x|$ events write on x; where $E_x \in \mathcal{P}(\varepsilon_0, \ldots e_{\mathsf{len}(v)})$ s.t. $\mathsf{conflict}_x(E_x) \in v$. Observe that for every event e_i s.t. $\varepsilon_i \in E_x$ and $\mathsf{suff}_x(\mathsf{v}_x, i)$ is arbitration-free, as CMod is closed under causal suffixes, there exists a visibility formula $v' \in \mathsf{CMod}$ s.t. $v'_x(e_i, e_{\mathsf{len}(v)})$. Thus, $|E_x| \geq \mathsf{af}(v)$, where $\mathsf{af}(v)$ is the number of arbitration-free suffixes of v. Moreover, as $e_0 \in \mathsf{rspec}(e_{\mathsf{len}(v)})(e_0, [\xi, \mathsf{CMod}])$, and v is not arbitration-free, $|E_x| > \mathsf{af}(v)$. However, as the layer bound of rspec, k, is smaller or equal than the number of arbitration-free suffixes of v, the number of events read by $f_{\mathsf{len}(v)}$ is at most $\mathsf{af}(v)$. Hence, $|E_x| \leq \mathsf{af}(v)$, which contradicts that $|E_x| > \mathsf{af}(v)$. We reach a contradiction; so the initial hypothesis, that v is non-vacuous w.r.t. CMod, is false. Thus, $v \notin \mathsf{CMod}$. □

5.12 Proof of the Arbitration-Free Consistency Theorem

Lemmas 5.12.1 and 5.10.2 prove the AFC theorem.

5.12.1 Arbitration-Freeness Implies Availability

The proof of (1) $\implies$ (2), essentially coincides with that of Lemma 5.6.6: we present an available Spec-implementation that guarantees $\mathtt{CC}$. As in Lemma 5.6.6, CMod is arbitration-free, so by Lemma 5.6.4, this implies that CMod is weaker than $\mathtt{CC}$. Thanks to Lemma 5.8.11, any implementation of $\mathtt{CC}$ also ensures CMod.

Lemma 5.12.1 ((1) $\implies$ (2)). *Let* OpSpec *be a basic operation specification. There exists an available* $(\mathtt{CC}, \mathsf{OpSpec})$*-implementation.*

Proof. The main difference in the construction w.r.t. the implementation shown in Lemma 5.6.6 corresponds to the transition function, Δ_{i}. More specifically, the main and only change arise in Equation (5.8), which is substituted by Equation (5.58).

$$\begin{aligned} M_t(e) &= [x \mapsto \mathsf{rspec}(e)(x, E^x_t(e))]_{x \in \mathsf{Keys}} \\ E^x_t(e) &= \left\{ e' \;\middle|\; \begin{array}{l} e' \in \mathsf{Events} \cap t \;\wedge\; e' \text{ writes } x \text{ in } \mathtt{exec}(t) \;\wedge \\ (\mathtt{rep}(e') = \mathtt{rep}(e) \vee \mathsf{rec}_t(e', e)) \end{array} \right\} \\ \mathsf{ar}^t_e &= \mathsf{ar}_{\restriction E^x_t(e) \times E^x_t(e)} \end{aligned} \tag{5.58}$$

Is immediate to show that I_E is a storage implementation. Showing that I_E is an available Spec-implementation is done as in Lemma 5.6.6. Observe that Lemmas 5.7.2 and 5.7.3 apply to this implementation; so $(S_{\mathsf{i}}, A_{\mathsf{i}}, s^{\mathsf{i}}_0, \Delta_{\mathsf{i}})$ is an available implementation. In Lemma 5.12.2 we show that indeed I_E is an implementation of $(\mathtt{CC}, \mathsf{Spec})$, concluding the result.

□

Lemma 5.12.2. *The implementation* I_E *is an implementation of* $\mathsf{Spec}' = (\mathtt{CC}, \mathsf{OpSpec})$.

Proof. Let $P_E = (S_{\mathsf{p}}, A_{\mathsf{p}}, s^{\mathsf{p}}_0, \Delta_{\mathsf{p}})$ be a program. We prove by induction on the length of all traces in $\mathcal{T}_{P_E \| I_E}$ that any trace t is valid w.r.t. Spec'. The base case, when t contains a single action, is immediate as such action corresponds to the initial event, which does not read any object. Let us assume that for any trace $t' \in \mathcal{T}_{P_E \| I_E}$ of at most length k, $\mathtt{exec}(t')$ is valid w.r.t. Spec'; and let us show that for any trace t of length $k+1$, $\mathtt{exec}(t)$ is also valid w.r.t. Spec'.

Let $h = (E, \mathsf{so}, \mathsf{wr})$ and $\xi = (h, \mathsf{rb}, \mathsf{ar})$ be respectively the history $\mathsf{history}(t)$ and the abstract execution $\mathtt{exec}(t)$. We denote sr to the induced order between send-receive actions with the same metadata on t. For proving that ξ is valid w.r.t. Spec', we first prove that ξ is indeed an abstract execution, i.e., ξ satisfies Definition 5.3.4. In particular, by the construction of $(S_{\mathsf{i}}, A_{\mathsf{i}}, s_0^{\mathsf{i}}, \Delta_{\mathsf{i}})$ (compared with that of Lemma 5.7.1), it suffices showing that $\mathsf{wr} \cup \mathsf{so} \subseteq \mathsf{rb}$.

By definition of rb, $\mathsf{so} \subseteq \mathsf{rb}$, so we focus on proving that $\mathsf{wr} \subseteq \mathsf{rb}$. Let w, r be events and x be an object s.t. $(w, r) \in \mathsf{wr}_x$. In such case, there is a pair of actions a_r, a_w s.t. $r \in a_r$, $w \in a_w$ and $w \in \mathsf{wr}\text{-}\mathtt{Set}(a_r)(x)$. Hence, $w \in \mathsf{rspec}(r)(x, E_t^x(r))$. We deduce then that $\mathsf{rec}_t(w, r)$ must hold; which implies that there exists a $\mathtt{send}$ action s and a $\mathtt{receive}$ action v s.t. $\mathsf{rb}\text{-}\mathtt{Set}(s) = \mathsf{rb}\text{-}\mathtt{Set}(v)$ and $w <_t s <_t v <_t r$. By $\mathsf{sendAllData}$ predicate, $w \in \mathsf{rb}\text{-}\mathtt{Set}(s)$; so by minRcv, $w \in \mathsf{rb}\text{-}\mathtt{Set}(v)$. By the induced abstract execution definition, $(w, r) \in \mathsf{rb}$.

Finally, we show that ξ is valid w.r.t. Spec'. By Definition 5.8.8, it suffices to show that for every event r and object x, $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, \mathsf{ctxt}_x(r, [\xi, \mathtt{CC}]))$. Let r be a read event, x be an object and $p = \mathsf{prefix}(t, r)$. We know by Equation (5.58) that $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, E_p^x(r))$. Observe that by Equation (5.58) and rb's definition, $E_p^x(r)$ coincides with $\mathsf{ctxt}_x(r, [t, \mathtt{CC}])$. Thus, so we conclude that $\mathsf{wr}_x^{-1}(r) = \mathsf{rspec}(r)(x, \mathsf{ctxt}_x(r, [t, \mathtt{CC}]))$. □

5.12.2 Availability Implies Arbitration-Freeness

The proof of this result mimics that of Lemma 5.6.7. We prove the contrapositive: if CMod is not arbitration-free, then no available Spec-implementation exists. Indeed, if CMod is not arbitration-free, every normal form CMod' of CMod contains a simple visibility formula involving ar, and such formula precludes the existence of an available $(\mathsf{CMod}, \mathsf{OpSpec})$-implementation (see Lemma 5.10.2).

Lemma 5.10.2. *Let* $\mathsf{Spec} = (\mathsf{CMod}, \mathsf{OpSpec})$ *be a storage specification. Assume that* CMod *contains a simple visibility formula* v *which is non-vacuous w.r.t.* OpSpec*, such that for some* $i, 0 \leq i \leq \mathsf{len}(\mathsf{v})$*,* $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$*. Then, there is no available* $(\mathsf{CMod}, \mathsf{OpSpec})$*-implementation.*

Proof. We assume by contradiction that there is an available implementation I_E of Spec but CMod contains a visibility formula v non-vacuous w.r.t. OpSpec s.t. for some $i, 0 \leq i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$. We use the latter fact to construct a specific program, which by the assumption, admits a trace (in the composition with this implementation) that contains no $\mathtt{receive}$ action. We show that any abstract execution induced by this trace, which is admissible by any available implementation of Spec, is not valid w.r.t. Spec. This contradicts the hypothesis.

The program P we construct generalizes the litmus program presented in Figure 5.1. P uses two replicas r_0, r_1, two distinguished objects x_0, x_1 and a collection of events $e_i^{x_l}, 0 \leq i \leq \mathsf{len}(v), l \in \{0, 1\}$. The events are used to "encode" two instances of v_{x_0} and v_{x_1}.

Let d_v be the largest index i s.t. $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$ (last occurrence of ar). Then, v is formed of two parts: the path of dependencies from ε_0 to ε_{d_v} which is not arbitration-free, and the suffix from ε_{d_v} up to $\varepsilon_{\mathsf{len}(v)}$, the arbitration-free part.

For ensuring that $\mathsf{v}_x(e_0^{x_l}, \ldots e_n^{x_l})$ holds in an induced abstract execution of a trace without $\mathtt{receive}$ actions, we require that if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, then $e_{i-1}^{x_l}$ is a write event and $e_i^{x_l}$ is a read event. For ensuring that $\mathsf{wrCons}_x^{\mathsf{v}}(e_0, \ldots e_{\mathsf{len}(v)})$ holds in such abstract execution, we consider a distinct object y_E, also distinct from x_0, x_1. These objects represents each different conflict in v in an explicit manner. Intuitively, we require that events $e_i^{x_l}$ write on object y_E (resp. x_l) iff $\varepsilon_i \in E$.

More formally, we denote by $E_x \in \mathcal{P}(\varepsilon_0, \dots e_{\mathsf{len}(v)})$ to the set s.t. $\mathsf{conflict}_x(E_x) \in v$. Also, for every $i, 0 \leq i \leq \mathsf{len}(v), l \in \{0, 1\}$, we denote by $X_i^{x_l}$ to the set containing objects y_E (resp. $\hat{x}_i^{x_l}$) iff $E \in \mathsf{conflictsOf(v, i)}$ (resp. $E_x \in \mathsf{conflictsOf(v, i)}$); where $\hat{x}_i^{x_l} = x_l$ if $i < d_v$ and x_{1-l} otherwise. We denote by X to the union of sets $X_i^{x_l}, 0 \leq i \leq \mathsf{len}(v), l \in \{0, 1\}$.

In the construction, we require that replica r_l executes events $e_i^{x_l}$ if $i < d_v$ and events $e_i^{x_{1-l}}$ otherwise – the replica r_l executes the non arbitration-free part of v for object x_l and the arbitration-free suffix of v for x_{1-l}. We denote by $r_i^{x_l}$ to such replica.

More in detail, we construct a set of events, E^i, histories, $h^i = (E^i, \mathsf{so}^i, \mathsf{wr}^i)$, and executions, $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$, $0 \leq i \leq \mathsf{len}(v)$ inductively, starting from an initial event `init`, and incorporating at each time a pair of new events, $e_i^{x_0}$ and $e_i^{x_1}$. We use the notation h^{-1} and ξ^{-1} to describe the history and abstract execution containing only `init` respectively. For simplifying notation, we use the convention $\texttt{init} = e_{-1}^{x_0} = e_{-1}^{x_1}$.

For the inductive step, we assume that the abstract execution $\xi^{i-1} = (h^{i-1}, \mathsf{rb}^{i-1}, \mathsf{ar}^{-1})$ associated to the history $h^{i-1} = (E^{i-1}, \mathsf{so}^{i-1}, \mathsf{wr}^{i-1})$ contains events $e_{-1}^{x_0} \dots e_{i-1}^{x_0}, e_{i-1}^{x_1}$ and is well-defined (satisfies Definition 5.3.4) and we construct the history h^i and the abstract execution ξ^i.

The construction of ξ^i follows the structure of that constructed in Lemma 5.6.7's proof, but with the technical details of that used in Theorem 5.11.9's proof.

For the inductive step, we assume that the abstract execution $\xi^{i-1} = (h^{i-1}, \mathsf{rb}^{i-1}, \mathsf{ar}^{-1})$ associated to the history $h^{i-1} = (E^{i-1}, \mathsf{so}^{i-1}, \mathsf{wr}^{i-1})$ contains events $e_{-1}^{x_0} \dots e_{i-1}^{x_0} e_{i-1}^{x_1}$ and is well-defined (satisfies Definition 5.3.4) and we construct the history h^i and the abstract execution ξ^i.

In the following, let $l \in \{0, 1\}$.

Like in Theorem 5.11.9, we define a pair of special objects, $\tilde{x}_i^{x_l}$ and $o_i^{x_l}$. The purpose of object $\tilde{x}_i^{x_l}$ is control the number of events in ξ that write object x_l. Equation (5.59) describes $\tilde{x}_i^{x_l}$; where choice is a function that deterministically chooses an element from a non-empty set. The object $o_i^{x_l}$ is an object different from objects $x, y_E, E \in \mathcal{P}(\varepsilon_0, \dots \varepsilon_{\mathsf{len}(v)})$ and $o_j^{x_{l'}}, -1 \leq j < i, l' \in \{0, 1\}$ that we use for ensuring that if $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, then $(e_{i-1}, e_i) \in \mathsf{wr}$. W.l.o.g., we can assume that $o_i^{x_0} \neq o_i^{x_1}$.

$$\tilde{x}_i^{x_l} = \begin{cases} \tilde{x}_{i-1}^{x_l} & \text{if } X_i^{x_l} = \emptyset \\ \hat{x}_i^{x_l} & \text{if } X_i^{x_l} \neq \emptyset \text{ and } \hat{x}_i^{x_l} \in X_i^{x_l} \\ \mathsf{choice}\,(X_i) & \text{if } X_i^{x_l} \neq \emptyset \text{ and } \hat{x}_i^{x_l} \notin X_i^{x_l} \end{cases} \tag{5.59}$$

We select a domain $D_i^{x_l}$, a set of objects $W_i^{x_l}, W_i^{x_l} \subseteq D_i^{x_l}$ that event $e_i^{x_l}$ must write, and a set of objects $C_i^{x_l} \subseteq D_i^{x_l}$ whose value needs to be corrected for events $e_i^{x_0}, e_i^{x_1}$ in ξ_{i+1} – in the sense of Definition 5.8.13. We distinguishing between several cases:

- $\underline{i = 0 \text{ or } 0 < i \leq \mathsf{len}(v) \text{ and } \mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr} \text{ and } \mathsf{conflictsOf(v, i)} \neq \emptyset}$: In this case, we select $e_i^{x_l}$ to be a write event. If OpSpec only allows single-object atomic read-write events, we define $D_i^{x_l} = X_i^{x_l}$; while if not, we consider a domain containing $o_{i-1}^{x_l}, o_i^{x_l}$, every object in $X_i^{x_l}$ but no object from $X \setminus X_i^{x_l}$ nor objects $o_j^{x_{l'}}, 0 \leq j < \mathsf{len}(v), l' \in \{0, 1\} j \neq i-1, i$. Observe that by Proposition 5.11.10, such domain always exist on OpSpec.

If there is an unconditional write event whose domain is $D_i^{x_l}$, we define $W_i^{x_l} = D_i^{x_l}$. Otherwise, we define $W_i^{x_l} = X_i^{x_l} \cup \{o_i^{x_l}\}$.

- $0 < i \leq \mathsf{len}(v)$, $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$ and $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset$: In this case, by Proposition 5.11.11, OpSpec allows atomic read-write events. If OpSpec only allows single-object atomic read-write events, we define $D_i^{x_l} = X_i^{x_l}$; while if not, we consider a domain containing o_{i-1}, o_i, every object in $X_i^{x_l}$ but no object from $X \setminus X_i^{x_l}$ nor objects $o_j, 0 \leq j < \mathsf{len}(v), j \neq i-1, i$. Observe that by Proposition 5.11.10, such domain always exist on OpSpec.

  Similarly to the previous case, if there is an unconditional atomic read-write event whose domain is $D_i^{x_l}$, we define $W_i^{x_l} = D_i^{x_l}$. Otherwise, we define $W_i^{x_l} = X_i^{x_l} \cup \{o_i^{x_l}\}$.

- $0 < i \leq \mathsf{len}(v)$ and $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) = \emptyset$: In this case, by Proposition 5.11.11, OpSpec allows events that do not unconditionally write. If OpSpec allows read events that are not write events, we select $D_i^{x_l}$ to be the domain of any such event and $W_i^{x_l} = \emptyset$. Otherwise, OpSpec must allow conditional write events; so we select $D_i^{x_l}$ to be the domain of any such event, $W_i^{x_l} = \emptyset$. Observe that in this case, thanks to the assumptions on OpSpec (see Section 5.8.4), we can assume without loss of generality that whenever $o_{i-1}^{x_l} \in D_{i-1}$, $o_{i-1}^{x_l} \in D_i^{x_l}$ as well; while otherwise, that $\tilde{x}_{i-1}^{x_l} \in D_i^{x_l}$.

Finally we describe the event $e_i^{x_l}$ thanks to the sets $D_i^{x_l}$ and $W_i^{x_l}$. If $W_i^{x_l} = D_i^{x_l}$ and $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, we select an unconditional atomic read-write event whose domain is $D_i^{x_l}$. If $W_i^{x_l} = D_i^{x_l}$ and $\mathsf{Rel}_i^{\mathsf{v}} \neq \mathsf{wr}$, we select an unconditional write event whose domain is $D_i^{x_l}$. If $W_i^{x_l} = \emptyset$ and OpSpec allows read events that are not write events, we select a read event whose domain is $D_i^{x_l}$. Finally, if that is not the case, we select a conditional write event $e_i^{x_l}$ s.t. $\mathsf{obj}(e_i^{x_l}) = D_i^{x_l}$ and s.t. an execution-corrector exists for $(e_i^{x_l}, W_i^{x_l}, \tilde{x}_i^{x_l}, \xi^{i-1} \oplus e_i^{x_0} \oplus e_i^{x_1})$. Such event always exists by the assumptions on operation specifications (Section 5.8.4). W.l.o.g. we can assume that $e_i^{x_l}$ happens on replica $r_i^{x_l}$.

For concluding the description of $h^i = (E_i, \mathsf{so}^i, \mathsf{wr}^i)$ and $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$, we use an auxiliary history and abstract execution, $h_{-1}^i = (E_{-1}^i, \mathsf{so}_{-1}^i, \mathsf{wr}_{-1}^i)$ and $\xi_{-1}^i = (h_{-1}^i, \mathsf{rb}_{-1}^i, \mathsf{ar}_{-1}^i)$ respectively. For specifying wr_{-1}^i, we define the context mapping $c^i : \mathsf{Keys} \to \mathsf{Contexts}$ in the same fashion as in Theorem 5.11.9:

$$c_i^{x_l}(y) = (F_i^{x_l}(y), \mathsf{rb}^{i-1}_{\restriction F_i^{x_l}(y) \times F_i^{x_l}(y)}, \mathsf{ar}^{i-1}_{\restriction F_i^{x_l}(y) \times F_i^{x_l}(y)}) \tag{5.60}$$

where $F_i^{x_l}(y)$ is the mapping associating each object y with the set of events described below:

$$F_i^{x_l}(y) = \begin{array}{ll} \{e \in E^{i-1} \mid \ \mathsf{wspec}(e)(y, [\xi^{i-1}, \mathsf{CC}]) \downarrow \text{and } (e, e_{i-1}^{x_{1-l}}) \in (\mathsf{rb}^{i-1})^* \ \} & \text{if } i = d_v \\ \{e \in E^{i-1} \mid \ \mathsf{wspec}(e)(y, [\xi^{i-1}, \mathsf{CC}]) \downarrow \text{and } (e, e_{i-1}^{x_l}) \in (\mathsf{rb}^{i-1})^* \ \} & \text{otherwise} \end{array}$$

Then, we define ξ_{-1}^i as the abstract execution of the history $h_{-1}^i = (E_{-1}^i, \mathsf{so}_{-1}^i, \mathsf{wr}_{-1}^i)$ obtained by appending $e_i^{x_0}, e_i^{x_1}$ to h_{-1}^i and ξ_{-1}^i as follows: E_{-1}^i contains E^{i-1} and events $e_i^{x_0}, e_i^{x_1}$. First of all, we require that the relations so_{-1}^i, wr_{-1}^i, rb_{-1}^i and ar_{-1}^i contain so^{i-1}, wr^{i-1}, rb^{i-1} and ar^{i-1} respectively.

With respect to events $e_i^{x_0}, e_i^{x_1}$, we impose that $e_i^{x_l}$ is the maximal event w.r.t. so_{-1}^i among those on the same replica. Also, $e_i^{x_l}$ is maximal w.r.t. wr as we define that for every object z, $((\mathsf{wr}_{-1}^i)_z)^{-1}(e_i^{x_l}) = \mathsf{rspec}(e_i^{x_l})(z, c_i^{x_l}(z))$. We also require that $\mathsf{rb}_{-1}^i = \mathsf{so}_{-1}^i$. With respect to ar_{-1}^i, we impose that $e_i^{x_0}$ succeeds every event in E^i w.r.t. ar_{-1}^i and that $e_i^{x_1}$ is the maximum event w.r.t. ar in ξ_{-1}^i.

We use ξ_{-1}^i to construct ξ^i. If event $e_i^{x_l}$ is not a conditional write event, $\xi^i = \xi_{-1}^i$. Otherwise, if event $e_i^{x_l}$ is a conditional write event, given $W_i^{x_l}$ and object $\tilde{x}_i^{x_l}$, we select an execution-corrector for $e_i^{x_l}$ w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$ and $a_i^{x_l}$. W.l.o.g., we assume that every event mapped by $a_i^{x_l}$ happens on replica $r_i^{x_l}$. Observe that by the choice of sets $D_i^{x_l}$ and $W_i^{x_l}$, and thanks to the assumptions on storages (see Section 5.8.4), such event(s) are always well-defined.

In addition, we denote by $C_i^{x_l}$ to the set of objects we need to correct for $e_i^{x_l}$. More specifically, if $e_i^{x_l}$ is a conditional write-read, we denote by $C_i^{x_l}$ to the set of objects y s.t. $a_i^{x_l}(y)$ is defined, i.e. $C_i^{x_l} = \{y \in \mathsf{Keys} \mid a_i^{x_l}(y) \downarrow\}$. In the case $e_i^{x_l}$ is not a conditional write-read, we use the convention $C_i^{x_l} = \emptyset$. The set of events in ξ^i is the following: $E^i = E^{i-1} \cup \bigcup_{l \in \{0,1\}} (\{e_i^{x_l}\} \bigcup_{y \in C_i \setminus \{o_{i-1}^{x_l}\}} a_i^{x_l}(y))$. Observe that by the choice of $C_i^{x_l}$, the set E_i is well-defined.

From ξ_{-1}^i, we define $\xi^i = \xi_0^i \overset{\mathsf{seq}(a_i)}{\curlyvee} e_i$ as the corrected execution of ξ and $e_i^{x_0}, e_i^{x_1}$ with events $a_i^{x_0}, a_i^{x_1}$. For describing ξ^i, we consider $<$ to be a well-founded order over Keys. ξ^i satisfies the following:

- $\underline{\mathsf{so}^i}$: Let $y \in C_i^{x_l}$. We require that for every event $e \in E^{i-1}$, $(e, a_i^{x_l}(y)) \in \mathsf{so}^i$ iff $\mathtt{rep}(e) = r_i^{x_l}, 0 \leq j < i$. We also require that $(\mathtt{init}, a_i^{x_l}(y)) \in \mathsf{so}^i$ and $(a_i^{x_l}(y), e_i^{x_l}) \in \mathsf{so}^i$. Finally, we require that for every objects $y' \in C_i^{x_l}, y' < y$, $(a_i^{x_l}(y'), a_i^{x_l}(y)) \in \mathsf{so}^i$.

- $\underline{\mathsf{wr}^i}$: Let y be an object in $C_i^{x_l}$. For every object z, if $z \in C_i^{x_l}$ and $z < y$, we require that $(\mathsf{wr}_z^i)^{-1}(a_i^{x_l}(y)) = \mathsf{rspec}(a_i^{x_l}(y))(z, c_i^{x_l}(z) \oplus a_i^{x_l}(z))$; while otherwise, we require that $(\mathsf{wr}_z^i)^{-1}(a_i^{x_l}(y)) = \mathsf{rspec}(a_i^{x_l}(y))(z, c_i^{x_l}(z))$. We also require that for every object z, if $z \in C_i^{x_l}$, then $(\mathsf{wr}_z^i)^{-1}(e_i^{x_l}) = \mathsf{rspec}(e_i^{x_l})(z, c_i^{x_l}(z) \oplus a_i^{x_l}(z))$, while otherwise, $(\mathsf{wr}_z^i)^{-1}(e_i^{x_l}) = \mathsf{rspec}(e_i^{x_l})(z, c_i^{x_l}(z))$.

- $\underline{\mathsf{rb}^i}$: Let $y \in C_i^{x_l}$. We require that for every object $y \in C_i^{x_l}$ and event e s.t. $(e, a_i^{x_l}(y)) \in \mathsf{so}^i$, s.t. $(e, a_i^{x_l}(y)) \in \mathsf{so}^i \cup \mathsf{wr}^i$, $(e, a_i^{x_l}(y)) \in \mathsf{rb}^i$.

- $\underline{\mathsf{ar}^i}$: We impose that for every event $e \in E^{i-1}$, $(e, a_i^{x_l}(y)) \in \mathsf{ar}^i, y \in C_i^{x_l}$. We also require that for every pair of objects $y_1, y_2 \in C_i$ s.t. y_1, y_2, $(a_i^{x_l}(y_1), a_i^{x_l}(y_2)) \in \mathsf{ar}^i$. As a tie-breaker between events associated to x_0 and x_1, we require that for every pair of events $e \in \{e_i^{x_0}, a_i^{x_0}(y) \mid y \in C_i^{x_0}\}$, $e' \in \{e_i^{x_1}, a_i^{x_1}(y) \mid y \in C_i^{x_0}\}$, $(e, e') \in \mathsf{ar}^i$.

We then define $h^i = (E^i, \mathsf{so}^i, \mathsf{wr}^i)$ and $\xi^i = (h^i, \mathsf{rb}^i, \mathsf{ar}^i)$. Observe that by construction of h^i and ξ^i, as $\mathsf{wr}^i \subseteq \mathsf{rb}^i = \mathsf{so}^i$, they satisfy Definitions 5.3.2 and 5.3.4 respectively; so they are a history and an abstract execution respectively. Also, observe that ξ^i is the corrected abstract execution of ξ_{-1}^i for events $e_i^{x_0}, e_i^{x_1}$ with events $a_i^{x_0}, a_i^{x_1}$, i.e. $\xi^i = \xi_1^i = \xi_0^i \overset{\mathsf{seq}(a_i^{x_1})}{\curlyvee} e_i^{x_1}$, where $\xi_0^i = \xi_{-1}^i \overset{\mathsf{seq}(a_i^{x_0})}{\curlyvee} e_i^{x_0}$.

Then, we define $\mathsf{Events}_\mathsf{p} = E^{\mathsf{len}(v)}$ as the set our program employs. The set $\mathsf{Events}_\mathsf{p}$ induces the set of traces $\mathcal{T}_\mathsf{p}$.

We define the program $P = (S_\mathsf{p}, A_\mathsf{p}, s_0^\mathsf{p}, \Delta_\mathsf{p})$, where $\mathtt{init}_\mathsf{p} = \mathtt{init}$ and Δ_p is the transition function defined as follows: for every trace $t \in \mathcal{T}_\mathsf{p}$ and event $e \in \mathsf{Events}_\mathsf{p}$, $\Delta_\mathsf{p}(t, e) \downarrow$ if and only if $e \not\in t$ and every event in $\mathsf{Events}_\mathsf{p}$ whose replica coincide with e and has smaller identifier than e is included in t.

The rest of the proof, which proceeds as follows, essentially combines previous results obtained while proving Lemma 5.6.7 and Theorem 5.11.9:

1. There exists a finite trace t of $P \parallel I_E$ that contains no receive action (Lemma 5.7.5)

2. The trace t induces a history $h_\mathsf{v} = (E, \mathsf{so}, \mathsf{wr})$ and an abstract execution $\xi_\mathsf{v} = (h, \mathsf{rb}, \mathsf{ar})$ where $\mathsf{rb} = \mathsf{so}$. As I_E is valid w.r.t. Spec, ξ_v is valid w.r.t. Spec. Next, we prove that since $\mathsf{rb} = \mathsf{so}$, events in ξ_v read the latests value w.r.t. so for any object. In particular, we deduce that ξ_v is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ (Corollary 5.12.5).

3. Since ar is a total order, either $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$ or $(e^{x_1}_{d_\mathsf{v}-1}, e^{x_0}_{d_\mathsf{v}-1}) \in \mathsf{ar}$. W.l.o.g., assume that $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$. By Proposition 5.12.6, we deduce that $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(\mathsf{v})}, [\xi_\mathsf{v}, \mathsf{CMod}])$. The proof is explained in Figure 5.9: if $(e^{x_0}_{d_\mathsf{v}-1}, e^{x_1}_{d_\mathsf{v}-1}) \in \mathsf{ar}$, then all events $e^{x_0}_i$ form a path in such way that $\mathsf{v}_{x_0}(e^{x_0}_0, \ldots e^{x_0}_{\mathsf{len}(\mathsf{v})})$ holds in ξ_v. If some event $e^{x_l}_i$ is a conditional read-write event, the predicate $\mathsf{conflict}_x(e^{x_0}_0, \ldots e^{x_0}_{\mathsf{len}(\mathsf{v})})$ holds in ξ_v thanks to the corrector events $A^{x_l}_i$.

4. As $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(v)}, [\xi_\mathsf{v}, \mathsf{CMod}])$ but $(e^{x_0}_0, e^{x_0}_{\mathsf{len}(\mathsf{v})}) \not\in \mathsf{rb}$ (no message is received), we deduce in Proposition 5.11.16 that OpSpec is layered w.r.t. ar. By contrapositive, if OpSpec would be layered w.r.t. rb, as $e^{x_0}_0 \in \mathsf{ctxt}_{x_0}(e^{x_0}_{\mathsf{len}(\mathsf{v})}, [\xi_\mathsf{v}, \mathsf{CMod}])$, there would exist an event e s.t. $(e^{x_0}_0, e) \in \mathsf{rb}$ and $e \in \mathsf{rspec}(e^{x_0}_{\mathsf{len}(\mathsf{v})})(x_0, [\xi_\mathsf{v}, \mathsf{CMod}])$. However, as $\mathsf{rb} = \mathsf{so}$, $\mathtt{rep}(e^{x_0}_0) = \mathtt{rep}(e) = \mathtt{rep}(e^{x_0}_{\mathsf{len}(\mathsf{v})})$ which is false because $\mathtt{rep}(e^{x_0}_0) = r_0$ and $\mathtt{rep}(e^{x_0}_{\mathsf{len}(\mathsf{v})}) = r_1$.

5. Since rspec is maximally layered, we can show that the layer bound of rspec is smaller than or equal to the number of arbitration-free suffixes of v (Proposition 5.11.17). Observe that an event writes x_0 only if it is $\mathtt{init}$ or is an event $e^{x_l}_i$ s.t. $\varepsilon_i \in E_x$ and $l = 0$. Any such index i corresponds to a suffix of v. By causal suffix closure, for any arbitration-free suffix v' of v there is a visibility formula that subsumes v' in $\mathsf{nCMod}_{\mathsf{OpSpec}}$. As d_v is the maximum index for which $\mathsf{Rel}^\mathsf{v}_i = \mathsf{ar}$, the number of events writing x_0 in replica r_1 distinct from $\mathtt{init}$ coincide with the number of arbitration-free suffixes of v. Hence, as rspec is layered w.r.t. ar, if its layer bound would be greater than the number of arbitration-free suffixes, $e^{x_0}_{\mathsf{len}(\mathsf{v})}$ would necessarily read x_0 from $\mathtt{init}$ (other events writing x_0 are in replica r_0 and $e_{\mathsf{len}(\mathsf{v})}$ only reads from events in r_1). However, as rspec is maximally-layered and $e^{x_0}_0$ succeeds $\mathtt{init}$ w.r.t. ar and rb^+, we would conclude that $e^{x_0}_{\mathsf{len}(\mathsf{v})}$ would also read x_0 from $e^{x_0}_0$. However, this is impossible as $\mathsf{wr} \subseteq \mathsf{rb} = \mathsf{so}$ but $e^{x_0}_0$ is in replica r_0 and $e^{x_0}_{\mathsf{len}(\mathsf{v})}$ is in replica r_1.

6. Lastly, we show in Proposition 5.11.18 that if the layer bound of rspec is smaller than or equal to the number of arbitration-free suffixes of v, then v is vacuous w.r.t. OpSpec, which contradicts the fact that v is a visibility formula from the normal form $\mathsf{nCMod}_{\mathsf{OpSpec}}$. □

Proposition 5.12.3. *The abstract execution $\xi^{\mathsf{len}(v)}$ described in Lemma 5.10.2 satisfies that for every $i, 0 \leq i \leq \mathsf{len}(v), l \in \{0,1\}$:*

1. *For every object $y \in C_i^{x_l}$, the following conditions hold:*
   (a) *For every object $z \in \mathsf{Keys}$, if $z \in C_i^{x_l}$ and $z < y$, $G(a_i^{x_l}(y), z) = F_i^{x_l}(z) \cup \{a_i^{x_l}(z)\}$, while otherwise, $G(a_i^{x_l}(y), z) = F_i^{x_l}(z)$.*
   (b) *The execution $\xi_l^i \upharpoonright y$ is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$.*
2. *For the event $e_i^{x_l}$, the following conditions hold:*
   (a) *For every object z, if $z \in C_i^{x_l}$, $G(e_i^{x_l}, z) = F_i^{x_l}(z) \cup \{a_i^{x_l}(z)\}$, while otherwise $G(e_i^{x_l}, z) = F_i^{x_l}(z)$.*
   (b) *The execution ξ_l^i is valid w.r.t. $(\mathsf{CC}, \mathsf{OpSpec})$.*

where $\mathsf{ctxt}_z(e, [\xi^{\mathsf{len}(v)}, \mathsf{CC}]) = (G(e,z), \mathsf{rb}_{\upharpoonright G(e,z) \times G(e,z)}, \mathsf{ar}_{\upharpoonright G(e,z) \times G(e,z)})$.

Proof. The proof of this result essentially coincides with that of Proposition 5.11.12.

We prove the result by induction. In particular, we show that for every $i, -1 \leq i \leq \mathsf{len}(v)$ and object y, either (0) $i = -1$ or (1) and (2) hold. The base case, $i = -1$, is immediate as (0) holds; so let us suppose that the result holds for every $j, -1 \leq j < i$, and let us prove it for i.

For proving the inductive step, we first prove (1) for $l = 0$, then (2) for $l = 0$, and then (1) and (2) for $l = 1$. As both (1) and (2) have an identical proof (observe that the role of object y in the former is just to declare that event $a_i^{x_l}(y)$ is well-defined and the role of l is to determine which session must be proven first), we present only the proof of (1) for $l = 0$.

We show (1) by transfinite induction. Let α be an ordinal of cardinality $|\mathsf{Keys}|$. For every $k, 0 \leq k \leq \alpha$, we denote by V_k to the set containing the first k elements in Keys according to $<$. We show that (1) holds for every $y \in V_k \cap C_i^{x_0}$.

The base, V_0 is immediate as $V_0 = \emptyset$. We thus focus on the successor case (i.e., showing that if (1) holds for every object $y \in V_k \cap C_i^{x_0}$ it also holds for V_{k+1}), as the limit case is immediate: if k is a limit ordinal, $V_k = \bigcup_{i, i<k} V_i$; so (1) immediately holds. For showing that (1) holds for every object $y \in V_{k+1} \cap C_i^{x_0}$, as by induction hypothesis it holds for every object $y \in V_k \cap C_i^{x_0}$, it suffices to show it for the only object $y \in V_{k+1} \setminus V_i$. W.l.o.g., we can assume that $y \in C_i^{x_0}$; as otherwise the result is immediate.

We first prove (1a) and then we show (1b). Let $z \in \mathsf{Keys}$ be an object. Two cases arise: $z \in C_i, z < y$ or not. Both cases are identical, so we present the former, i.e., if $z \in C_i^{x_0}, z < y$, then $F_i^{x_0}(z) \cup \{a_i^{x_0}(z)\} = G(a_i^{x_0}(y), z)$.

For proving that $F_i^{x_0}(z) \cup \{a_i^{x_0}(z)\} \subseteq G(a_i^{x_0}(y), z)$, we distinguish whether $i = d_v$ or not. However, the proof essentially coincides in both cases, so we present the case $i = d_v$.

We split the proof in two blocks: showing that $F_i^{x_0}(z) \subseteq G(a_i^{x_0}(y), z)$ and showing that $a_i^{x_0}(z) \in G(a_i(y), z)$.

For showing that $F_i^{x_0}(z) \subseteq G(a_i^{x_0}(y), z)$, let e be an event in $F_i^{x_0}(z)$. In such case, $e \in E^{i-1}$, $\mathsf{wspec}(e)(z, [\xi^i, \mathtt{CC}]) \downarrow$ and $(e, e_{i-1}^{x_1}) \in (\mathsf{rb}^i)^*$. By the construction of ξ, it is easy to see that any such event belongs to E^i, $\mathsf{wspec}(e)(z, [\xi, \mathtt{CC}]) \downarrow$ and $(e, e_{i-1}^{x_1}) \in (\mathsf{rb}^{\mathsf{len}(v)})^*$. As $i = d_v$, we deduce that $(e_{i-1}^{x_1}, a_i^{x_0}(y)) \in \mathsf{rb}^i \subseteq \mathsf{rb}^{\mathsf{len}(v)}$. Hence, $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^{\mathsf{len}(v)})^+$; so $e \in G(a_i^{x_0}(y), z)$. This show that $F_i^{x_0}(z) \subseteq G(a_i^{x_0}(y), z)$.

For showing that $a_i^{x_0}(z) \in G(a_i^{x_0}(y), z)$, we observe that $\xi_0^i = \xi_{-1}^i \overset{a_i^{x_0}}{\curlyvee} e_i^{x_0}$. We note that as $z < y$, by induction hypothesis (1b), $\xi_0^i \upharpoonright z$ is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$. Thus, by Property 1 of Definition 5.8.13, $\mathsf{wspec}(a_i(z))(z, [\xi_0^i, \mathtt{CC}]) \downarrow$. Hence, $\mathsf{wspec}(a_i^{x_0}(z))(z, [\xi^i, \mathtt{CC}]) \downarrow$ and $\mathsf{wspec}(a_i^{x_0}(z))(z, [\xi^{\mathsf{len}(v)}, \mathtt{CC}]) \downarrow$. As $z < y$, $(a_i^{x_0}(z), a_i^{x_0}(y)) \in \mathsf{so}^i \subseteq \mathsf{so}^{\mathsf{len}(v)}$; so we conclude that $a_i^{x_0}(z) \in G(a_i^{x_0}(y), z)$.

We conclude the proof of the inductive step of (1a) by showing the converse i.e. $F_i^{x_0}(z) \cup \{a_i^{x_0}(z)\} \supseteq G(a_i^{x_0}(y), z)$. Let $e \in G(a_i^{x_0}(y), z)$. First of all, by the definition of Causal visibility formula (see Figure 5.4b), $e \in G(a_i^{x_0}(y), z)$ iff $\mathsf{wspec}(e)(z, [\xi, \mathtt{CC}]) \downarrow$ and $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^{\mathsf{len}(v)})^+$. Observe that if $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^{\mathsf{len}(v)})^+$, by construction of $\xi^{\mathsf{len}(v)}$, such event must belong to E^i, $\mathsf{wspec}(e)(z, [\xi^i, \mathtt{CC}]) \downarrow$ and $(e, a_i(y)) \in (\mathsf{rb}^i)^+$. We prove that if $e \in E^{i-1}$ then $e \in F_i^{x_0}(z)$, while otherwise, if $e \in E^i \setminus E^{i-1}$, then $e = a_i^{x_0}(z)$.

If $e \in E^{i-1}$, as $\mathsf{wspec}(e)(z, [\xi^i, \mathtt{CC}]) \downarrow$, $\mathsf{wspec}(e)(z, [\xi^{i-1}, \mathtt{CC}]) \downarrow$. Also, as $i = d_v$ and $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^i)^+$, we deduce that $(e, e_{i-1}^{x_1}) \in (\mathsf{rb}^{i-1})^*$. In other words, $e \in F_i^{x_0}(z)$.

Otherwise, if $e \in E^i \setminus E^{i-1}$, we note that by construction of $\xi^{\mathsf{len}(v)}$, the only events in $E^i \setminus E^{i-1}$ s.t. $(e, a_i^{x_0}(y)) \in (\mathsf{rb}^i)^+$ are events $a_i^{x_0}(w), w \in C_i, w < y$. As $\xi_0^i = \xi_{-1}^i \overset{\mathsf{seq}(a_i^{x_0})}{\curlyvee} e_i^{x_0}$ and $z < y$, $\xi_0^i \upharpoonright z$ is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$. Hence, $\mathsf{wspec}(e)(z, [\xi^i, \mathtt{CC}]) \downarrow$ iff $\mathsf{wspec}(e)(z, [\xi_0^i, \mathtt{CC}]) \downarrow$. Thus, by Property 1 of Definition 5.8.13 we conclude that $e = a_i^{x_0}(z)$.

For concluding the inductive step, we show that (1b) holds. This is immediate by the definition of wr^i: for every event $e \in \xi^i \upharpoonright y$, by induction hypothesis (1a) or (2a) – depending on whether $e = e_j^{x_{l'}}$ or $a_j^{x_{l'}}(w)$, where $0 \leq j \leq i, w \in C_i, l' \in \{0, 1\}$ – $(\mathsf{wr}^i)_z^{-1}(e) = \mathsf{rspec}(e)(\mathtt{CC}, [\xi_{l'}^j \upharpoonright y, z]) = \mathsf{rspec}(e)(\mathtt{CC}, [\xi_l^i \upharpoonright y, z])$. Thus, $\xi^i \upharpoonright y$ is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$. $\square$

A consequence of Proposition 5.12.3 is the following result.

Corollary 5.12.4. *The abstract execution ξ described in Lemma 5.10.2 is valid w.r.t.* $(\mathtt{CC}, \mathsf{OpSpec})$.

Corollary 5.12.5 is an immediate result from Corollary 5.12.4, obtained by simply observing that $\mathsf{rb}^{\mathsf{len}(v)} = \mathsf{so}^{\mathsf{len}(v)} = \mathsf{so} = \mathsf{rb}$.

Corollary 5.12.5. *The abstract execution ξ_v described in Lemma 5.10.2 is valid w.r.t.* $(\mathtt{CC}, \mathsf{OpSpec})$.

Proposition 5.12.6. *For every $l \in \{0, 1\}$, if $(e_{d_v-1}^{x_l}, e_{d_v-1}^{x_{1-l}}) \in \mathsf{ar}$, then the predicate $v_{x_0}(e_0^{x_l}, \dots e_{\mathsf{len}(v)}^{x_l})$ holds in the abstract execution $\xi = (h, \mathsf{rb}, \mathsf{ar})$ described in Theorem 5.11.9.*

Proof. The proof of this result essentially coincides with that of Proposition 5.11.14.

The proof is a simple consequence of $\xi^{\mathsf{len}(v)}$'s construction. To show that $v_{x_0}(e_0^{x_l}, \dots e_{\mathsf{len}(v)}^{x_l})$ holds in ξ, we first show that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{Rel}_i^{\mathsf{v}}$ and to then prove that $\mathsf{wrCons}_x^{\mathsf{v}}(e_0^{x_l}, \dots e_{\mathsf{len}(v)}^{x_l})$ holds in ξ.

We prove that for every $i, 1 \leq i \leq \mathsf{len}(v)$, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{Rel}_i^{\mathsf{v}}$. Four cases arise depending on $\mathsf{Rel}_i^{\mathsf{v}}$.

- $\underline{\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{so}}$: In this case, by construction of events $e_{i-1}^{x_l}, e_i^{x_l}$, we know that $r_i^{x_l} = r_{i-1}^{x_l}$. Hence, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{so}^i \subseteq \mathsf{so}$.
- $\underline{\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}}$: In this case, we first show that there is an object $y \in D_i^{x_l} \cap W_{i-1}^{x_l} \setminus C_i^{x_l}$, and then show that $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{wr}_y$. For showing the first part, we distinguish between cases depending on whether $o_{i-1}^{x_l} \in D_i^{x_l}$ or not.
  - $\underline{o_{i-1}^{x_l} \in D_i^{x_l}}$: In this sub-case, we show that $y = o_{i-1}^{x_l}$. On one hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) = \emptyset$, by the choice of event $e_i^{x_l}$, $o_{i-1}^{x_l} \in D_{i-1}^{x_l} \setminus C_i^{x_l}$. On the other hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset$, as $o_{i-1}^{x_l} \in D_i^{x_l}$, we deduce that OpSpec allows multi-object read-write events. Observe that as v is conflict-maximal, $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}-1) \neq \emptyset$. Hence, as OpSpec allows multi-object read-write events, we deduce that $o_{i-1}^{x_l} \in D_{i-1}^{x_l} \setminus C_i^{x_l}$. In both cases, as $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}-1) \neq \emptyset$ and $o_{i-1}^{x_l} \in D_{i-1}^{x_l}$, by the choice of $W_{i-1}^{x_l}$, we conclude that $o_{i-1}^{x_l} \in W_{i-1}^{x_l}$.
  - $\underline{o_{i-1}^{x_l} \notin D_i^{x_l}}$: In this case, we show that $y = \tilde{x}_i^{x_l}$. On one hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) = \emptyset$, $X_i^{x_l} = \emptyset$; so by the choice of $\tilde{x}_i^{x_l}$ (see Equation (5.59)), $\tilde{x}_i^{x_l} = \tilde{x}_{i-1}^{x_l}$. By the choice of $D_i^{x_l}$, $\tilde{x}_{i-1}^{x_l} \in D_i^{x_l} \setminus C_i^{x_l}$. Moreover, as v is conflict-maximal, $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}-1) \neq \emptyset$; so $\tilde{x}_{i-1}^{x_l} \in X_{i-1}^{x_l}$. By the choice of event $e_{i-1}^{x_l}$, $X_{i-1}^{x_l} \subseteq W_{i-1}^{x_l}$. Altogether, we conclude that $\tilde{x}_i^{x_l} \in W_{i-1}^{x_l}$.

    On the other hand, if $\mathsf{conflictsOf}(\mathsf{v}, \mathsf{i}) \neq \emptyset$, we note that $\tilde{x}_i^{x_l} \in D_i^{x_l} \setminus C_i^{x_l}$. As $o_{i-1}^{x_l} \notin D_i^{x_l}$, we deduce that OpSpec only allows single-object read-write events. Thus, $D_i^{x_l} = \{\tilde{x}_i^{x_l}\}$. As v is conflict-maximal, we deduce that $X_i^{x_l} \subseteq X_{i-1}^{x_l}$. As by the choice of $e_{i-1}^{x_l}$, $X_{i-1}^{x_l} \subseteq W_{i-1}^{x_l}$, we conclude that $\tilde{x}_i^{x_l} \in W_{i-1}^{x_l}$.

  We prove now that $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{wr}_y$. First, we show that $e_{i-1}^{x_l}$ writes y in ξ. On one hand, if $e_{i-1}^{x_l}$ is an unconditional write event, $\mathsf{wspec}(e_{i-1}^{x_l})(y, c_i^{x_l}(y)) \downarrow$. On the other hand, if $e_{i-1}^{x_l}$ is a conditional write event, as ξ is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ (Corollary 5.12.5) and $y \in W_i^{x_l}$, by Property 2 of Definition 5.8.13, we deduce that $\mathsf{wspec}(e_{i-1}^{x_l})(y, c_i^{x_l}(y)) \downarrow$. Then, as $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{wr}$, $i \neq d_v$, so $e_{i-1}^{x_l} \in F_i^{x_l}(y)$. Observe that by construction of ξ, $e_{i-1}^{x_l}$ is the *so*-maximum event in $c_i^{x_l}(y)$. As every event in $F_i^{x_l}(y)$ is so-related, we deduce that $e_{i-1}^{x_l}$ is the ar-maximum event in $F_i^{x_l}(y)$. We note that as $y \notin C_i^{x_l}$, by Proposition 5.12.3, $F_i^{x_l}(y) = G(e_i^{x_l}, y)$. Altogether, $e_{i-1}^{x_l}$ is the ar-maximum event in $\mathsf{ctxt}_y(e_i^{x_l}, [\xi^{\mathsf{len}(v)}, \mathtt{CC}])$. As $\mathsf{rb}^{\mathsf{len}(v)} = \mathsf{rb}$, we conclude that $e_{i-1}^{x_l}$ is the ar-maximum event in $\mathsf{ctxt}_y(e_i^{x_l}, [\xi, \mathtt{CC}])$. As rspec is maximally layered, we deduce that $e_{i-1}^{x_l} \in \mathsf{rspec}(e_i^{x_l})(y, [\xi, \mathtt{CC}])$. Finally, as ξ is valid w.r.t. $\mathtt{CC}$ (Corollary 5.12.5), we conclude that $(e_{i-1}^{x_l}, e_i) \in \mathsf{wr}_y$.

- $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{rb}$: In this case, $i \neq d_v$. Then, $\mathsf{rb} = \mathsf{so}$ and $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{so}$, we conclude that $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{rb}$.
- $\mathsf{Rel}_i^{\mathsf{v}} = \mathsf{ar}$: On one hand, if $i = d_v$, by hypothesis, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{ar}$. On the other hand, if $i \neq d_v$, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{so}$. Thus, $(e_{i-1}^{x_l}, e_i^{x_l}) \in \mathsf{ar}$.

For showing that show that $\mathsf{wrCons}_x^{\mathsf{v}}(e_0, \dots e_{\mathsf{len}(v)})$, we show that for every $i, 0 \leq i \leq \mathsf{len}(v)$ and every set $E \in \mathsf{conflictsOf}(\mathsf{v}, \mathsf{i})$, the event $e_i^{x_l}$ writes on object y_E[9]. If $e_i^{x_l}$ is an unconditional write, by the choice of $e_i^{x_l}$, it writes on every object in $D_i^{x_l}$. As $y_E \in D_i^{x_l}$, we conclude that $e_i^{x_l}$ writes on y_E. Otherwise, if $e_i^{x_l}$ is a conditional write, we observe that $y_E \in W_i^{x_l}$. Hence, as $\xi_0^i = \xi_{-1}^i \overset{\mathsf{seq}(a_i^{x_0})}{\curlyvee} e_i^{x_0}$ and ξ_0^i is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$ (resp. $\xi_1^i = \xi_0^i \overset{\mathsf{seq}(a_i^{x_1})}{\curlyvee} e_i^{x_1}$ and ξ_1^i is valid w.r.t. $(\mathtt{CC}, \mathsf{OpSpec})$) (Proposition 5.12.3), we deduce using Property 2 of Definition 5.8.13 that $\mathsf{wspec}(e_i^{x_l})(y_E, [\xi^i, \mathtt{CC}]) \downarrow$. By construction of ξ, we conclude that $\mathsf{wspec}(e_i^{x_l})(y_E, [\xi, \mathtt{CC}]) \downarrow$. $\square$

5.13 Related Work and Discussion

The CAP conjecture [36] claims that a distributed key-value store cannot be both consistent, available and tolerate partitions. The proof of the *CAP theorem* [59], uses a so-called *split brain* behavior, where two sets of replicas are isolated from each other, and a *get* (read) operation misses the result of an *earlier set* (write) operation (which completes before the get starts). We remark that our proof in section 5.2 actually extends the proof of the CAP theorem so it holds without the real-time requirement used in the original proof [59].

As pointed by some critiques of the CAP theorem (e.g., [70]), the proof equates consistency with atomicity of read / write variables. Moreover, network partitioning is a stand-in for end-to-end delays in geo-distributed systems. The PACELC (*if Partition then Availability or Consistency, Else Latency or Consistency*) theorem [3] (see [62]) captures these observations; its proof extends results proved for *sequential consistency* [78, 18]. These results are also proved for read / write variables, capturing key-value stores. In the executions we construct, messages between replicas are delayed, in a way that corresponds to split brain behavior, and our emphasis is on constructing the right interaction sequences. We believe this behavior can be used to extend the AFC theorem so it talks about latency, rather than availability, for the same interaction sequences.

The CALM (*consistency as logical monotonicity*) conjecture [65] relates monotonicity of queries to lack of coordination. Informally, it states that a query has a coordination-free execution strategy if and only if it is monotonic. In order to make this statement more concrete, it is necessary do define what coordination freedom means. In their proof of the CALM theorem, Ameloot et al. [15] equate coordination-freedom with the ability of clients to produce an output even when there is no communication between replicas. The proof relies on a split brain behavior, somewhat similar to the one used in the CAP theorem [59]. Extensions of this theorem [16] equip replicas with knowledge of the data distribution. The CALM theorem is motivated, in part, by Bloom [14], a programming language that encourages order-insensitive programming. The applications they present are to key-value stores and

[9]For simplifying the proof, we abuse of notation and say that $y_E = x_l$ if $E = E_x$. Observe that v is conflict-maximal, either $\mathsf{conflict}_x(E_x)$ or $\mathsf{conflict}(E_x)$ do not belong to v.

to a shopping cart, essentially, a counter. Later work extends the CALM approach to a programming environment for composing small *lattices* [14], and relates it to CRDTs [74].

One key challenge in deriving our result is considering abstract, generic consistency models, while prior work considers specific models. The other challenge is to allow their composition with abstract, generic shared objects, while prior work mostly consider key-value stores. On the possibility side, this is facilitated by the relating arbitration-freeness to causality; the necessity side relies on finding carefully-designed client interactions that “stress” dependencies between replicas.

Defining available implementations for causal consistency has been considered in several works [21, 80, 82, 22]. The work of Attiya, Ellen, and Morrison [19] and Mahajan, Alvisi, Dahlin, et al. [83] show that, in the case of multi-value registers, consistency models stronger than causal consistency cannot support available implementations. In [19] the condition is *observable causal consistency* (OCC) whereas in [83] the condition is *real-time causal consistency* (RTC). The definition of both OCC and RTC are specific to multi-value registers, and the impossibility result depends on several restrictions that we do not consider. Both papers make some (nontrivial, but different) assumptions about the implementations. Furthermore, both of them do not truly prove a tight result: while both [19, 83] prove the positive result for CC, in [19], the impossibility is proved for OCC, and in [83] it is for RTC (both stronger than CC). Besides handling a more general class of operations, the AFC theorem is a strengthening of their results, as it applies to causal consistency and is therefore tight.

Our specification framework builds on previous work [40, 29, 41, 45]. Similarly to Burckhardt et al. [41, 40], storage system specifications decouple consistency from the object semantics. We re-use the same ideas of defining consistency using visibility formulas, contexts, and an arbitration relation. Our object semantics is split into several semantical functions (rspec, extract, and wspec) in order to be more general (modeling transactions), and be able to express “normal” constrains. The extension to transaction isolation levels is similar to Cerone, Bernardi, and Gotsman [45] and Biswas and Enea [29].

6 | Conclusions and Future Work

In this thesis, we provided a better understanding of distributed storage systems under different consistency models. We now give a brief summary of the content presented in this work and conclude with future research lines.

Conclusions

Reasoning about and correctly implementing weak consistency models presents significant challenges. In Chapter 2, we presented an axiomatic semantics of weak consistency models, inspired on the work of [29] that can provide a better understanding of isolation levels to developers than previous work [9].

In Chapter 3, we presented the first DPOR SMC model-checking algorithm for checking consistency on distributed key-value database with static set of keys under weak isolation levels. We presented a generic class of DPOR SMC algorithms, called *swapping-based* algorithms, and discussed EXPLORE-CE, a swapping based algorithm that is a sound, complete, strongly optimal algorithm and employs polynomial memory for causally extensible isolation levels (including Transactional Causal Consistency, Read Atomic and Read Committed isolation levels). We also proved that no swapping based algorithm with those four properties exists for Snapshot Isolation and Serializability. We then presented a optimal but not strongly optimal variant of EXPLORE-CE for Snapshot Isolation and Serializability and evaluated its performance on benchmarks from the literature.

In Chapter 4, we presented an axiomatic semantics for transactional SQL programs. For describing behaviors of a program, we introduced the notion of *client* and *full* histories. We studied the complexity of checking if a history is consistent w.r.t. its associated isolation configuration. We showed that for full histories the results from [29] easily translate, while for client histories the problem is in general NP-complete. This shows that handling SQL-like semantics is strictly more complex than the standard read-write semantics. We also presented an algorithm for checking consistency of a client history that is exponential on worst-case scenarios but polynomial-time on relevant cases. We evaluated our algorithm on benchmarks from the literature.

Finally, in Chapter 5 we proved the AFC theorem, i.e. that storage specifications have an available implementation if and only if their consistency model is arbitration-free. For that, we described a framework for defining consistency models that builds on [29], as well as a framework for specifying operation semantics inspired by the work of [41]. It is the first result of its kind taking into account both consistency constraints and the semantics of the implemented storage.

Future Work

In the follow-up work, we would like to explore some the following open questions.

- Among the isolation levels considered in [45, 29], those that can be checked in polynomial time on static set of keys are exactly those that are causally extensible (see Section 3.3.2). Whether such characterization can be extended to arbitrary isolation levels, for example those defined as in Section 4.3, remains unknown.

- In [84], the authors explore a variant of TruSt algorithm [73] for mixed-size accesses, and claim that the approach can be extended to the transactional case on databases. While their approach is still not strongly optimal, the authors claim that the number of explored inconsistent executions can be bounded with respect to the number of concurrent clients of the program. However, the authors present an algorithm that explore consistent executions, unlike Algorithm 1 that reduce the state space by exploring histories. Whether both approaches can be combined for minimizing the impact of not being able to obtain strong optimality while retaining the benefits of history-based algorithms is not yet studied.

- Algorithm 1 is a DPOR SMC algorithm designed only for databases with static sets of keys. Analyzed histories in this context can be reinterpreted under SQL as full histories with specific `WHERE` clauses only querying one key at a time. Seems reasonable to assume that Algorithm 1 can be easily adapted to full histories with SQL-like transactions.

  However, as discussed in Section 3.7, the bottleneck of Algorithm 1 is the number of explored histories. In general, multiple full histories are witnesses of the same client history; and in comparison with Algorithm 1, the running time of Algorithm 8 are almost negligible (see Section 4.6). It remains unknown whether Algorithm 1 can also be extended to the case of client histories and its scalability in such case.

- Algorithms 7 and 8 can be only applied to some specific set of isolation levels. In particular, they cannot handle Transactional Causal Consistency. How can they be adapted to handle other isolation levels it is still not explored.

- The running time of Algorithm 7 depends on the number of possible consistent prefixes. Such exploration determines the commit order used for checking consistency. However, it is not always necessary computing a total order for checking consistency. For example, in the case of heterogeneous isolation configuration with only k transactions running Serializability, we can deduce consistency based on k partial orders, that are total orders when restricted to the k Serializable transactions. The complexity analysis of an algorithm with respect to some external parameter k is a classic question from the parametrized complexity field. Whether the benefits of such field can help us to improve our algorithms, both theoretically and effectively, remains an open question.

- The AFC theorem (Theorem 5.10.1) is incomparable with other well-known results such as [19]. The constraints imposed on the storage specifications do not apply on those. It remains unexplored whether relaxing our conditions for accepting a wider range of storage specifications, without substantially change the result, may subsume the aforementioned work.

- The extension of the CALM theorem to CRDTs [74] suggest that it can also be extended to arbitrary storage systems. Our AFC theorem shows that coordination-free implementations of storage specifications should enforce an arbitration-free consistency model. However, it is unclear how monotonicity translates to arbitrary operation specifications, and whether CALM theorem can be extended to storage specifications with consistency models weaker than Causal Consistency.

Bibliography

[1] Ansi x3.135-1992. Technical report, American National Standards Institute, November 1992. URL https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[2] *Consistency Levels in Azure Cosmos DB*, 2020. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[3] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, 2012. doi: 10.1109/MC.2012.33. URL https://doi.org/10.1109/MC.2012.33.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 134–156. Springer, 2016. doi: 10.1007/978-3-319-41540-6_8. URL https://doi.org/10.1007/978-3-319-41540-6_8.

[5] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. *Acta Informatica*, 54(8):789–818, 2017. doi: 10.1007/s00236-016-0275-0. URL https://doi.org/10.1007/s00236-016-0275-0.

[6] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017. doi: 10.1145/3073408. URL https://doi.org/10.1145/3073408.

[7] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA):135:1–135:29, 2018. doi: 10.1145/3276505. URL https://doi.org/10.1145/3276505.

[8] 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):150:1–150:29, 2019. doi: 10.1145/3360576. URL https://doi.org/10.1145/3360576.

[9] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD thesis, 1999.

[10] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In David B. Lomet and Gerhard Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78. IEEE Computer Society, 2000. doi: 10.1109/ICDE.2000.839388. URL https://doi.org/10.1109/ICDE.2000.839388.

[11] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless model checking under a reads-value-from equivalence. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 341–366. Springer, 2021. doi: 10.1007/978-3-030-81685-8_16. URL https://doi.org/10.1007/978-3-030-81685-8_16.

[12] Deepthi Devaki Akkoorath and Annette Bieniusa. Antidote: the highly-available geo-replicated database with strongest guarantees. Technical report, 2016. URL https://pages.lip6.fr/syncfree/attachments/article/59/antidote-white-paper.pdf.

[13] Peter Alvaro and Kyle Kingsbury. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, 2020. doi: 10.5555/3430915.3442427. URL http://www.vldb.org/pvldb/vol14/p268-alvaro.pdf.

[14] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260. www.cidrdb.org, 2011. URL http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf.

[15] Tom J. Ameloot, Frank Neven, and Jan Van Den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2), May 2013. doi: 10.1145/2450142.2450151. URL https://doi.org/10.1145/2450142.2450151.

[16] Tom J. Ameloot, Bas Ketsman, Frank Neven, and Daniel Zinn. Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the calm-conjecture. *ACM Trans. Database Syst.*, 40(4), December 2015. doi: 10.1145/2809784. URL https://doi.org/10.1145/2809784.

[17] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April*

14-20, 2018, Proceedings, Part II, volume 10806 of *Lecture Notes in Computer Science*, pages 229–248. Springer, 2018. doi: 10.1007/978-3-319-89963-3_14. URL https://doi.org/10.1007/978-3-319-89963-3_14.

[18] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994. doi: 10.1145/176575.176576. URL https://doi.org/10.1145/176575.176576.

[19] Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):141–155, January 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2556669. URL https://doi.org/10.1109/TPDS.2016.2556669.

[20] Hagit Attiya, Constantin Enea, and Enrique Román-Calvo. Arbitration-free consistency is available (and vice versa). *CoRR*, abs/2510.21304, 2025. doi: 10.48550/ARXIV.2510.21304. URL https://doi.org/10.48550/arXiv.2510.21304.

[21] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, 2012. URL http://doi.acm.org/10.1145/2391229.2391251.

[22] 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*, pages 761–772, 2013. URL http://doi.acm.org/10.1145/2463676.2465279.

[23] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In Laurent Réveillère, Tim Harris, and Maurice Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 6:1–6:16. ACM, 2015. doi: 10.1145/2741948.2741972. URL https://doi.org/10.1145/2741948.2741972.

[24] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Checking robustness against snapshot isolation. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 286–304. Springer, 2019. doi: 10.1007/978-3-030-25543-5_17. URL https://doi.org/10.1007/978-3-030-25543-5_17.

[25] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Robustness against transactional causal consistency. In Wan J. Fokkink and Rob van Glabbeek, editors, *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*, volume 140 of *LIPIcs*, pages 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi: 10.4230/LIPIcs.CONCUR.2019.30. URL https://doi.org/10.4230/LIPIcs.CONCUR.2019.30.

[26] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):110, May 1995. ISSN 0163-5808. doi: 10.1145/568271.223785. URL https://doi.org/10.1145/568271.223785.

[27] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995. doi: 10.1145/223784.223785. URL https://doi.org/10.1145/223784.223785.

[28] Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.CONCUR.2016.7. URL https://doi.org/10.4230/LIPIcs.CONCUR.2016.7.

[29] Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA):165:1–165:28, 2019. doi: 10.1145/3360591. URL https://doi.org/10.1145/3360591.

[30] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. Monkeydb: effectively testing correctness under weak isolation levels. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021. doi: 10.1145/3485546. URL https://doi.org/10.1145/3485546.

[31] 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. doi: 10.1145/3009837.3009888. URL https://doi.org/10.1145/3009837.3009888.

[32] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. Dynamic partial order reduction for checking correctness against transaction isolation levels. *Proc. ACM Program. Lang.*, 7(PLDI):565–590, 2023. doi: 10.1145/3591243. URL https://doi.org/10.1145/3591243.

[33] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. Artifact for "on the complexity of checking mixed isolation levels for sql transactions", October 2024. URL https://github.com/Galieve/benchbase-histories.

[34] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. On the complexity of checking mixed isolation levels for SQL transactions. In Ruzica Piskac and Zvonimir Rakamaric, editors, *Computer Aided Verification - 37th International Conference,*

CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part IV, volume 15934 of *Lecture Notes in Computer Science*, pages 315–337. Springer, 2025. doi: 10.1007/978-3-031-98685-7_15. URL https://doi.org/10.1007/978-3-031-98685-7_15.

[35] Martin Brain, James H. Davenport, and Alberto Griggio. Benchmarking solvers, sat-style. In Matthew England and Vijay Ganesh, editors, *Proceedings of the 2nd International Workshop on Satisfiability Checking and Symbolic Computation co-located with the 42nd International Symposium on Symbolic and Algebraic Computation (ISSAC 2017), Kaiserslautern, Germany, July 29, 2017*, volume 1974 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL http://ceur-ws.org/Vol-1974/RP3.pdf.

[36] Eric A. Brewer. Towards robust distributed systems (invited talk). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, New York, NY, USA, 2000. ISBN 1581131836. URL https://doi.org/10.1145/343477.343502.

[37] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: facebook's distributed data store for the social graph. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 49–60. USENIX Association, 2013. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

[38] Lucas Brutschy, Dimitar K. Dimitrov, Peter Müller, and Martin T. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. 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 458–472. ACM, 2017. doi: 10.1145/3009837.3009895. URL https://doi.org/10.1145/3009837.3009895.

[39] Lucas Brutschy, Dimitar K. Dimitrov, Peter Müller, and Martin T. Vechev. Static serializability analysis for causal consistency. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 90–104. ACM, 2018. doi: 10.1145/3192366.3192415. URL https://doi.org/10.1145/3192366.3192415.

[40] Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, 2014. doi: 10.1561/2500000011. URL https://doi.org/10.1561/2500000011.

[41] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *41st Symposium on Principles of Programming Languages, POPL*, pages 271–284. ACM, 2014. URL https://doi.org/10.1145/2535838.2535848.

[42] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 568–590. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPICS.ECOOP.2015.568. URL `https://doi.org/10.4230/LIPIcs.ECOOP.2015.568`.

[43] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distributed Syst.*, 16(7):663–671, 2005. doi: 10.1109/TPDS.2005.86. URL `https://doi.org/10.1109/TPDS.2005.86`.

[44] Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. *J. ACM*, 65(2): 11:1–11:41, 2018. doi: 10.1145/3152396. URL `https://doi.org/10.1145/3152396`.

[45] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory, CONCUR*, pages 58–71, 2015. doi: 10.4230/LIPICS.CONCUR.2015.58. URL `https://doi.org/10.4230/LIPIcs.CONCUR.2015.58`.

[46] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Data-centric dynamic partial order reduction. *Proc. ACM Program. Lang.*, 2(POPL):31:1–31:30, 2018. doi: 10.1145/3158119. URL `https://doi.org/10.1145/3158119`.

[47] Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. Value-centric dynamic partial order reduction. *Proc. ACM Program. Lang.*, 3(OOPSLA):124:1–124:29, 2019. doi: 10.1145/3360550. URL `https://doi.org/10.1145/3360550`.

[48] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum, editors, *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 117–126. ACM Press, 1983. doi: 10.1145/567067.567080. URL `https://doi.org/10.1145/567067.567080`.

[49] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *Int. J. Softw. Tools Technol. Transf.*, 2(3):279–287, 1999. doi: 10.1007/s100090050035. URL `https://doi.org/10.1007/s100090050035`.

[50] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium*

on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007, pages 205–220. ACM, 2007. doi: 10.1145/1294261.1294281. URL https://doi.org/10.1145/1294261.1294281.

[51] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, 2013. doi: 10.14778/2732240.2732246. URL http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

[52] Michael Emmi and Constantin Enea. Sound, complete, and tractable linearizability monitoring for concurrent collections. *Proc. ACM Program. Lang.*, 2(POPL):25:1–25:27, 2018. doi: 10.1145/3158113. URL https://doi.org/10.1145/3158113.

[53] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30 (2):492–528, 2005. doi: 10.1145/1071610.1071615. URL https://doi.org/10.1145/1071610.1071615.

[54] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005. doi: 10.1145/1040305.1040315. URL https://doi.org/10.1145/1040305.1040315.

[55] Florian Furbach, Roland Meyer, Klaus Schneider, and Maximilian Senftleben. Memory-model-aware testing: A unified complexity analysis. *ACM Trans. Embed. Comput. Syst.*, 14(4):63:1–63:25, 2015. doi: 10.1145/2753761. URL https://doi.org/10.1145/2753761.

[56] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. Isodiff: Debugging anomalies caused by weak isolation. *Proc. VLDB Endow.*, 13(12):27732786, July 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407860. URL https://doi.org/10.14778/3407790.3407860.

[57] Phillip B. Gibbons and Ephraim Korach. On testing cache-coherent shared memories. In Lawrence Snyder and Charles E. Leiserson, editors, *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '94, Cape May, New Jersey, USA, June 27-29, 1994*, pages 177–188. ACM, 1994. doi: 10.1145/181014.181328. URL https://doi.org/10.1145/181014.181328.

[58] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997. doi: 10.1137/S0097539794279614. URL https://doi.org/10.1137/S0097539794279614.

[59] 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. ISSN

0163-5700. doi: 10.1145/564585.564601. URL https://doi.org/10.1145/564585.564601.

[60] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. ISBN 3-540-60761-7. doi: 10.1007/3-540-60761-7. URL https://doi.org/10.1007/3-540-60761-7.

[61] Patrice Godefroid. Model checking for programming languages using verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 174–186. ACM Press, 1997. doi: 10.1145/263699.263717. URL https://doi.org/10.1145/263699.263717.

[62] Wojciech M. Golab. Proving PACELC. *SIGACT News*, 49(1):73–81, 2018. doi: 10.1145/3197406.3197420. URL https://doi.org/10.1145/3197406.3197420.

[63] Alex Gontmakher, Sergey V. Polyakov, and Assaf Schuster. Complexity of verifying java shared memory execution. *Parallel Process. Lett.*, 13(4):721–733, 2003. doi: 10.1142/S0129626403001628. URL https://doi.org/10.1142/S0129626403001628.

[64] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 371–384. ACM, 2016. doi: 10.1145/2837614.2837625. URL https://doi.org/10.1145/2837614.2837625.

[65] Joseph M. Hellerstein. The declarative imperative: Experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39(1):5–19, September 2010. doi: 10.1145/1860702.1860704. URL https://doi.org/10.1145/1860702.1860704.

[66] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: A raft-based HTAP database. *Proc. VLDB Endow.*, 13(12):3072–3084, 2020. doi: 10.14778/3415478.3415535. URL http://www.vldb.org/pvldb/vol13/p3072-huang.pdf.

[67] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. Efficient black-box checking of snapshot isolation in databases. *Proc. VLDB Endow.*, 16(6):1264–1276, 2023. doi: 10.14778/3583140.3583145. URL https://www.vldb.org/pvldb/vol16/p1264-wei.pdf.

[68] Sudhir Jorwekar, Alan D. Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In Christoph Koch, Johannes Gehrke,

Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1263–1274. ACM, 2007. URL http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf.

[69] Gowtham Kaki, Kapil Earanky, K. C. Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2 (OOPSLA):164:1–164:27, 2018. doi: 10.1145/3276534. URL https://doi.org/10.1145/3276534.

[70] Martin Kleppmann. A critique of the CAP theorem. *CoRR*, abs/1509.05393, 2015. URL http://arxiv.org/abs/1509.05393.

[71] Michalis Kokologiannakis and Viktor Vafeiadis. HMC: model checking for hardware memory models. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1157–1171. ACM, 2020. doi: 10.1145/3373376.3378480. URL https://doi.org/10.1145/3373376.3378480.

[72] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 96–110. ACM, 2019. doi: 10.1145/3314221.3314609. URL https://doi.org/10.1145/3314221.3314609.

[73] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proc. ACM Program. Lang.*, 6 (POPL):1–28, 2022. doi: 10.1145/3498711. URL https://doi.org/10.1145/3498711.

[74] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M. Hellerstein. Keep CALM and CRDT on. *Proc. VLDB Endow.*, 16(4): 856–863, 2022. doi: 10.14778/3574245.3574268. URL https://www.vldb.org/pvldb/vol16/p856-power.pdf.

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

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

[77] Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In

Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 281–292. USENIX Association, 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

[78] Richard J Lipton and Jonathan S Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton University, Department of Computer Science, August 1988.

[79] Si Liu, Long Gu, Hengfeng Wei, and David A. Basin. Plume: Efficient and complete black-box checking of weak isolation levels. *Proc. ACM Program. Lang.*, 8(OOPSLA2): 876–904, 2024. doi: 10.1145/3689742. URL https://doi.org/10.1145/3689742.

[80] 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 23rd ACM Symposium on Operating Systems Principles*, pages 401–416, 2011. URL http://doi.acm.org/10.1145/2043556.2043593.

[81] 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 Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 401–416. ACM, 2011. doi: 10.1145/2043556.2043593. URL https://doi.org/10.1145/2043556.2043593.

[82] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 313–328. USENIX Association, 2013. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd.

[83] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. Consistency, availability, and convergence. *University of Texas at Austin Tech Report*, 11:158, 2011.

[84] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. Model checking C/C++ with mixed-size accesses. *Proc. ACM Program. Lang.*, 9(POPL):2232–2252, 2025. doi: 10.1145/3704911. URL https://doi.org/10.1145/3704911.

[85] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986. doi: 10.1007/3-540-17906-2_30. URL https://doi.org/10.1007/3-540-17906-2_30.

[86] Kartik Nagar and Suresh Jagannathan. Automated detection of serializability violations under weak consistency. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 41:1–41:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPIcs.CONCUR.2018.41. URL https://doi.org/10.4230/LIPIcs.CONCUR.2018.41.

[87] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 544–571. Springer, 2020. doi: 10.1007/978-3-030-44914-8_20. URL https://doi.org/10.1007/978-3-030-44914-8_20.

[88] Brian Norris and Brian Demsky. Cdschecker: checking concurrent data structures written with C/C++ atomics. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 131–150. ACM, 2013. doi: 10.1145/2509136.2509514. URL https://doi.org/10.1145/2509136.2509514.

[89] Burcu Kulahcioglu Ozkan. Verifying weakly consistent transactional programs using symbolic execution. In Chryssis Georgiou and Rupak Majumdar, editors, *Networked Systems - 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3-5, 2020, Proceedings*, volume 12129 of *Lecture Notes in Computer Science*, pages 261–278. Springer, 2020. doi: 10.1007/978-3-030-67087-0_17. URL https://doi.org/10.1007/978-3-030-67087-0_17.

[90] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979. doi: 10.1145/322154.322158. URL https://doi.org/10.1145/322154.322158.

[91] Andrew Pavlo. What are we doing with our lives?: Nobody cares about our concurrency control research. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, page 3. ACM, 2017. doi: 10.1145/3035918.3056096. URL https://doi.org/10.1145/3035918.3056096.

[92] Jos Rolando Guay Paz. *Microsoft Azure Cosmos DB Revealed: A Multi-Modal Database Designed for the Cloud*. Apress, USA, 1st edition, 2018. ISBN 1484233506.

[93] Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International*

Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993. doi: 10.1007/3-540-56922-7_34. URL https://doi.org/10.1007/3-540-56922-7_34.

[94] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982. doi: 10.1007/3-540-11494-7_22. URL https://doi.org/10.1007/3-540-11494-7_22.

[95] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. CLOTHO: directed test generation for weakly consistent database systems. *Proc. ACM Program. Lang.*, 3(OOPSLA):117:1–117:28, 2019. doi: 10.1145/3360543. URL https://doi.org/10.1145/3360543.

[96] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems SSS*, volume 6976, pages 386–400. Springer, 2011. doi: 10.1007/978-3-642-24550-3_29. URL https://doi.org/10.1007/978-3-642-24550-3_29.

[97] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 413–424. ACM, 2015. doi: 10.1145/2737924.2737981. URL https://doi.org/10.1145/2737924.2737981.

[98] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400. ACM, 2011. doi: 10.1145/2043556.2043592. URL https://doi.org/10.1145/2043556.2043592.

[99] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed SQL database. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1493–1509. ACM, 2020. doi: 10.1145/3318464.3386134. URL https://doi.org/10.1145/3318464.3386134.

[100] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In

Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994, pages 140–149. IEEE Computer Society, 1994. doi: 10.1109/PDIS.1994.331722. URL https://doi.org/10.1109/PDIS.1994.331722.

[101] TPC. Technical report, Transaction Processing Performance Council, February 2010. URL http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[102] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989. doi: 10.1007/3-540-53863-1_36. URL https://doi.org/10.1007/3-540-53863-1_36.

[103] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In George S. Avrunin and Gregg Rothermel, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, pages 97–107. ACM, 2004. doi: 10.1145/1007512.1007526. URL https://doi.org/10.1145/1007512.1007526.

[104] Todd Warszawski and Peter Bailis. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 520, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3064037. URL https://doi.org/10.1145/3035918.3064037.

[105] ANSI X3. 135-1992. american national standard for information systems-database language-sql. Technical report, 1992.