Université Paris Cité
ED 386 : Sciences Mathématiques de Paris Centre
Institut de Recherche en Informatique Fondamentale (IRIF)

Formal Verification of Concurrent Data Structures

par
Berk Cirisci

Thèse de doctorat d'informatique

dirigée par
Constantin Enea

Présentée et soutenue publiquement le 15 Decembre 2022
Devant un jury composé de :

Noam Rinetzky	*Associate Professor*	Tel Aviv University	Rapporteur
Thomas Wies	*Associate Professor*	New York University	Rapporteur
Andreas Podelski	*Professeur*	University of Freiburg	Examinateur
Mihaela Sighirenau	*Professeure*	ENS Cachan	Examinatrice
Ahmed Bouajjani	*Professeur*	Université Paris Cité, IRIF	Examinateur
Constantin Enea	*Professeur*	LIX, Ecole Polytechnique	Directeur

Abstract

Modern automated services rely on concurrent software where multiple requests are processed by different processes at the same time. These processes execute concurrently either in a single machine or in a distributed system with multiple machines built on top of a network. The requests are typically about processing data which is stored in data structures that provide implementations of common abstract data types (ADTs) such as queues, key-value stores, and sets. The data structures themselves are concurrent in the sense that they support operations that can execute concurrently at the same time.

Developing such concurrent data structures or even understanding and reasoning about them can be tricky. This is coming from the fact that synchronization between operations of these data structures must be minimized to reduce response time and increase throughput, yet this minimal amount of synchronization must also be adequate to ensure conformance to their specification. These opposing statements, along with the general challenge in reasoning about interleavings between operations make concurrent data structures a ripe source of insidious programming errors that are difficult to reproduce, locate, or fix. Therefore, verification techniques that can check the correctness of a concurrent data structure or (if it is not correct) that can detect, pinpoint and fix the errors in it, are invaluable.

In this thesis, we introduce new algorithmic approaches for improving the reliability of concurrent data structures. For shared-memory data structures (where processes communicate using a shared memory), we introduce new algorithms for finding safety violations (if any) and repairing the implementation in order to exclude these violations. For data structures running on top of a network, we focus on the underlying consensus protocols and provide a new proof methodology for checking their safety specification which is based on refinement.

Keywords: Formal Verification, Concurrent Data Structures, Linearizability, Model Checking, Partial Order Reduction, Consensus Protocols, Refinement

Résumé

Les services automatisés modernes reposent sur des logiciels concurrents où plusieurs demandes sont traitées par différents processus en même temps. Ces processus s'exécutent simultanément sur une seule machine ou dans un système distribué avec plusieurs machines reliés par un réseau. Les demandes concernent généralement le traitement de données stockées dans des structures de données qui fournissent des implémentations de types de données abstraits (ADT) courants tels que des files d'attente, des tables clé-valeur et des ensembles. Les structures de données elles-mêmes sont concurrentes dans le sens où elles permettent des opérations qui peuvent s'exécuter simultanément.

Développer de telles structures de données concurrentes ou même les comprendre et raisonner dessus peut être difficile. Cela vient du fait que la synchronisation entre les opérations de ces structures de données doit être minimisée pour réduire le temps de réponse et augmenter le débit, mais cette synchronisation doit également être adéquate pour assurer la conformité à leur spécification. Ces contraintes qui s'opposent, ainsi que le défi général du raisonnement sur les entrelacements entre les opérations, font des structures de données concurrentes une source d'erreurs de programmation insidieuses, difficiles à reproduire, localiser ou corriger. Par conséquent, les techniques de vérification qui peuvent vérifier la correction d'une structure de données concurrente ou (si elle n'est pas correcte) qui peuvent détecter, identifier et corriger les erreurs qu'elle contient, sont très importantes.

Dans cette thèse, nous introduisons de nouvelles approches algorithmiques pour améliorer la fiabilité des structures de données concurrentes. Pour les structures de données à mémoire partagée (où les processus communiquent à l'aide d'une mémoire partagée), nous introduisons de nouveaux algorithmes pour trouver les violations de sûreté (le cas échéant) et réparer l'implémentation afin d'exclure ces violations. Pour les structures de données qui m'exécutent au-dessus d'un réseau, nous nous concentrons sur les protocoles de consensus sous-jacents et nous fournissons une nouvelle méthodologie de preuve basée sur le raffinement pour vérifier leur spécification de sûreté.

Mots-clés: Vérification formelle, Structures de données concurrentes, Linéarisabilité, Model Checking, Réduction d'ordre partielle, Protocoles de consensus, Raffinement

Résumé

Les services automatisés modernes reposent sur des logiciels concurrents où plusieurs demandes sont traitées par différents processus en même temps. Ces processus s'exécutent simultanément sur une seule machine ou dans un système distribué avec plusieurs machines reliés par un réseau. Les demandes concernent généralement le traitement de données stockées dans des structures de données qui fournissent des implémentations de types de données abstraits (ADT) courants tels que des files d'attente, des tables clé-valeur et des ensembles. Les structures de données elles-mêmes sont concurrentes dans le sens où elles permettent des opérations qui peuvent s'exécuter simultanément.

Développer de telles structures de données concurrentes ou même les comprendre et raisonner dessus peut être difficile. Cela vient du fait que la synchronisation entre les opérations de ces structures de données doit être minimisée pour réduire le temps de réponse et augmenter le débit, mais cette synchronisation doit également être adéquate pour assurer la conformité à leur spécification. Ces contraintes qui s'opposent, ainsi que le défi général du raisonnement sur les entrelacements entre les opérations, font des structures de données concurrentes une source d'erreurs de programmation insidieuses, difficiles à reproduire, localiser ou corriger. Par conséquent, les techniques de vérification qui peuvent vérifier la correction d'une structure de données concurrente ou (si elle n'est pas correcte) qui peuvent détecter, identifier et corriger les erreurs qu'elle contient, sont très importantes.

Dans cette thèse, nous introduisons de nouvelles approches algorithmiques pour améliorer la fiabilité des structures de données concurrentes. Pour les structures de données à mémoire partagée (où les processus communiquent à l'aide d'une mémoire partagée), nous introduisons de nouveaux algorithmes pour trouver les violations de sûreté (le cas échéant) et réparer l'implémentation afin d'exclure ces violations. Pour les structures de données qui m'exécutent au-dessus d'un réseau, nous nous concentrons sur les protocoles de consensus sous-jacents et nous fournissons une nouvelle méthodologie de preuve basée sur le raffinement pour vérifier leur spécification de sûreté.

1 Vérification de propriétés de sûreté pour des programmes à mémoire partagée

1.1 Énoncé du problème

Notre premier objectif est de vérifier les structures de données concurrentes qui s'exécutent au-dessus d'une mémoire partagée. Le critère de correction standard dans ce cas est la linéarisabilité. Nous nous intéressons au problème de la vérification de la linéarisabilité d'une structure de données concurrente donnée.

Ce problème peut être vu comme une instance d'un problème plus général qui consiste à vérifier une propriété de sûreté pour un programme concurrent donné. Le programme concurrent est un client de la structure de données, qui appelle ses méthodes à partir d'un certain nombre (éventuellement illimité) de threads.

1.2 L'approche

Nous allons considérer une approche qui consiste à appliquer la vérification de modèle à un client borné constitué d'un nombre borné de threads, chaque thread effectuant un nombre borné d'appels. La vérification de modèle [24, 25, 84] explore l'espace d'états d'un programme donné de manière systématique et vérifie que chaque état atteignable satisfait une propriété donnée. Il fournit une couverture élevée des comportements du programme, mais il souffre du célèbre problème d'explosion d'état, c'est-à-dire que le nombre d'entrelacements possibles augmente de manière exponentielle dans la taille du code source. Pour résoudre le problème d'explosion d'état, nous utiliserons la réduction d'ordre partiel (POR) [24, 40, 41, 44, 51, 82, 93] qui limite le nombre d'entrelacements explorés sans sacrifier la couverture.

1.3 L'état de l'art

L'ensemble des entrelacements explorés par une technique POR est défini en restreignant l'ensemble des transitions explorées à partir de chaque état (point d'ordonnancement). Selon la classe de spécifications, les hypothèses sur les programmes ou les objectifs d'optimalité, il existe différentes définitions pour cet ensemble de threads, y compris les sleep sets [41], stubborn sets[41, 44, 93], persistent sets [40, 43, 60], ample sets [24, 51] et source sets [3].

La conception d'un algorithme de vérification de modèle basé sur POR doit prendre en compte plusieurs compromis de calcul. Premièrement, un tel algorithme peut sauvegarder les états explorés dans le passé ou pas [39], ce qui correspond à un compromis entre consommation mémoire et temps d'exécution. Deuxièmement, le calcul de l'ensemble des threads explorés à partir d'un certain état peut être plus ou moins complexe. Se concentrer sur l'optimalité théorique, c'est-à-

dire explorer exactement un entrelacement de chaque trace de Mazurkiewicz, peut rendre ce calcul plus complexe. Troisièmement, les algorithmes POR peuvent calculer les informations qu'ils utilisent à des fins de réduction de manière statique, par une analyse statique du code source, ou dynamiquement, lors de l'exploration des entrelacements. Le calcul statique est généralement moins cher et moins précis que le calcul dynamique.

1.4 Nos contributions

Nous proposons plusieurs algorithmes de vérification de modèle basés sur POR qui sauvegardent les états explorés dans le passé, en mettant l'accent sur la performance globale plutôt que sur l'optimalité théorique. Ces algorithmes combinent des calculs statiques et dynamiques des ensembles de threads à explorer à partir d'un état donné, dont la définition est basée sur les source sets. Nous avons montré que la performance des algorithmes dépend principalement du potentiel de réduction dans l'espace d'état. Nous avons identifié des utilisations souhaitables de ces algorithmes en fonction des ressources disponibles. Nous avons également analysé les effets potentiels de différentes stratégies de recherche (énumérations de pas d'exécution).

2 Identifier la cause fondamentale des violations de linéarisabilité

2.1 Énoncé du problème

Nous considérons le problème du débogage des violations de linéarisabilité, en trouvant leur cause fondamentale et suggérant des réparations possibles. Il s'agit d'un problème difficile car les violations de linéarisabilité peuvent contenir un large nombre de pas de calcul et imbriquer les calculs de différents threads de manière très compliqué.

Nous pouvons aborder ce problème en trois étapes. Tout d'abord, nous devrions définir une certaine notion de cause fondamentale pour la non-linéarisabilité. Deuxièmement, nous devons proposer des algorithmes efficaces pour trouver la cause fondamentale d'une violation. Et comme troisième étape, nous devons fournir une méthodologie efficace pour supprimer la violation qui a cette cause fondamentale.

2.2 L'approche

L'approche que nous avons choisie pour identifier les causes fondamentales des erreurs de linéarisabilité consiste à trouver des blocs de code (réparations) dans le code du programme dont l'atomicité est requise pour exclure toutes les exécutions non linéarisables données. L'idée clé de cette approche est que le problème peut être réduit à un problème algorithmique plus simple d'identification

de causes fondamentales minimales des violations de la sérialisabilité des conflits dans une exécution erronée. Dans un programme concurrent avec une décomposition donnée de son code en blocs de code, une exécution satisfait la sérialisabilité des conflits si elle est équivalente à une exécution dans laquelle tous les blocs de code sont exécutés de manière séquentielle non entrelacée. Une réparation peut être défini en utilisant une décomposition de l'ensemble de pas d'une exécution en un ensemble de blocs appelés intervalles, de sorte que cette exécution ne satisfait pas la sérialisabilité des conflits par rapport à cette décomposition. Ces intervalles correspondront à des sections atomiques qui seront recommandées comme réparations en utilisant notre approche.

2.3 L'état de l'art

Il existe différents travaux tels que [22, 54, 55, 56, 57, 59, 81, 99] pour la localisation de fautes, l'explication des erreurs, la minimisation de contre-exemples et la synthèse des bogues pour les programmes concurrents. Un autre ensemble de travaux [8, 15, 17, 23, 48, 73, 94, 95] concerne la correction des erreurs de concurrence en utilisant des techniques de synthèse de synchronisation qui peuvent aider les développeurs à réparer ces erreurs. À notre connaissance, Flint [70] est la seule autre approche qui se concentre sur le calcul de causes fondamentales des violations de linéarisabilité. Cependant, il se concentre sur un seul type de données abstrait, les tables clé-valeur.

2.4 Nos contributions

Dans cette thèse, nous présentons une nouvelle approche qui diagnostique les causes fondamentales des erreurs de linéarisabilité. Nous calculons les réparations optimales où l'optimalité fait référence au fait de permettre un nombre maximal d'entrelacements. Cependant, il peut y avoir plusieurs réparations optimales et nous devons déterminer laquelle est la plus susceptible de correspondre à la cause fondamentale. Par conséquent, nous définissons une heuristique pour classer ces réparations optimales en privilégiant celles qui désactivent le moin d'exécutions linéarisables. Nous proposons des résultats théoriques décrivant la réduction du problème principal à un problème lié aux violations de la sérialisabilité des conflits, et un algorithme polynomial pour résoudre ce dernier. Nous avons implémenté notre approche et mené plusieurs expériences en utilisant des structures de données concurrentes réalistes pour démontrer son efficacité.

3 Vérification de la sûreté des protocoles de consensus

3.1 Énoncé du problème

La mise en œuvre de structures de données concurrentes linéarisables qui fonctionnent sur un réseau repose généralement sur des algorithmes utilisés pour résoudre un problème plus générique

appelé la réplication de machine d'état. En général, les protocoles de réplication de machine d'état font partie d'une classe plus large de protocoles appelés protocoles de consensus. Les protocoles de consensus sont construits pour se mettre d'accord sur une seule valeur (décret unique) ou plusieurs valeurs ordonnées (multi-décret). Développer des implémentations pratiques correctes de protocoles de consensus ou même raisonner sur leur correction est très difficile. Leur complexité découle des différentes hypothèses sur l'environnement dans lequel ils fonctionnent (c'est-à-dire, défaillances de processus ou de liaison réseau, défaillances byzantines, etc.). Dans cette thèse, nous considérons le problème du raisonnement sur la sûreté des protocols de consensus.

3.2 L'approche

Pour raisonner sur la sûreté des protocoles de consensus, notre approche est basée sur le raffinement. Cela consiste à montrer que les comportements d'un protocole donné peuvent être mis en correspondance avec les comportements d'une spécification opérationnelle abstraite (définie comme une machine à états) dont la sûreté a déjà été prouvée. Ce mapping doit garantir que la sûreté de la spécification implique la sûreté du protocole d'origine. Concevoir de telles abstractions peut être très utile pour maîtriser la complexité lors de la conception de nouveaux protocoles, ou pour gagner en confiance quant à leur correction sans passer par des arguments de preuve ad hoc et fragiles comme dans les preuves de sûreté basées sur des invariants inductifs. Néanmoins, trouver le bon compromis entre l'expressivité d'une abstraction et la "facilité" de la relier à des protocoles concrets est une question importante et non triviale en général.

3.3 L'état de l'art

Le problème de prouver la correction des protocoles de consensus a été étudié dans la literature. Il existe des travaux qui utilisent des invariants inductifs [78, 79], qui sont cependant difficiles à inventer puisqu'ils doivent décrire tous les états intermédiaires produits par tous les ordres possibles de réception des messages. Certains autres travaux [6, 9, 36, 37, 38, 49, 63, 102] sont basés sur le raffinement. La plupart de ces travaux [36, 37, 38, 49, 63, 102] utilisent le raffinement pour prouver un seul protocole spécifique et les abstractions qu'ils dérivent au cours de ce processus ne se généralisent pas au-delà de ce protocole. Quelques-uns d'entre eux comme [6, 9] ont étudié des abstractions plus génériques mais lier un protocole concret à de telles abstractions est assez lourd et complexe.

3.4 Nos contributions

Nous proposons une nouvelle représentation abstraite de la dynamique des protocoles de consensus qui se concentre sur les quorums de réponses (votes) à une demande (proposition) qui se

forment au cours d'une exécution du protocole. Cette abstraction, formalisée sous la forme d'un objet arbre séquentiel appelé QTree, se ressemble à la description des protocoles récents utilisés dans les infrastructures Blockchain, par exemple, le protocole supportant Bitcoin ou Hotstuff. Nous montrons que cette abstraction peut être utilisée pour raisonner de manière uniforme sur la sûreté de divers algorithmes, par exemple, Paxos, PBFT, Raft et HotStuff. En général, elle fournit un nouvel argument basé sur l'induction pour prouver que de tels protocoles sont sûrs.

Contents

1 Introduction

In today's world, the *Internet* has become essential as most industries are growing online, and this does not seem to be changing in the near future. Each new day, more people are using web-based applications and their demands from them are increasing. To handle this high demand, the systems are *concurrent* in the sense that multiple requests are executed by different *processes* at the same time. These requests can be executed concurrently either in a single machine or in a *distributed* system with multiple machines built on top of a *network*. Since these requests are typically about processing data, one must consider how these requests interfere with each other. For instance one process can try to read a piece of data that is being deleted in parallel by some other process or some group of processes may fail arbitrarily. This data is typically stored in some *data structure* that provide implementations of common abstract data types (ADTs) such as queues, key-value stores, and sets. For processing the data concurrently, also these data structures must be concurrent in the way that they need to support operations to be done at the same time. Hence, concurrent systems are dependent on *reliable concurrent data structures*.

Developing such data structures or even understanding and reasoning about them can be tricky. This is coming from the fact that synchronization between operations of these data structures must be minimized to reduce response time and increase throughput, yet this minimal amount of synchronization must also be adequate to ensure that programs satisfy their specification. These opposing statements, along with the general challenge in reasoning about interleavings between operations make concurrent data structures a ripe source of insidious programming errors that are difficult to reproduce, locate, or fix. Therefore, verification techniques that can check the correctness of a concurrent data structure or (if it is not correct) that can detect, pinpoint and fix the errors in it, are invaluable.

Implementations of concurrent data structures either assume that processes communicate with each other using a *shared memory* or *message passing*. Generally speaking, shared memory is used between processes of a single machine system or between the threads in the same process in which processes *read and write* to common set of locations. On the other hand, message passing is used in a distributed system in which processes communicate through the network. When the processes execute requests that are transferred via a network, they must agree on the total order be-

tween these requests. This is achieved using complicated consensus protocols where the correctness is an infamously difficult problem.

In shared memory systems, one must reason about the possible interleavings between read and write operations to identify which of them are safe to run in parallel. To restrict the additional interleavings between the operations, synchronization primitives such as semaphores or monitors are used. As expected, using more synchronization primitives decreases the number of interleavings and thus, it is easier to reason about them. However, usage of these primitives deteriorates the level of concurrency and the performance. That's why implementations of concurrent data structures use these primitives in a very limited way and they become highly complex.

In message passing systems, processes have their own local memories and communicate through the network by exchanging (sending and receiving) messages. These messages invoke processes to take some actions as long as the predefined conditions in their source codes are satisfied. Messages can be transferred with either *synchronous or asynchronous message passing*. In synchronous message passing, sender processes remain busy until their requests are finalized by the receiver processes. On the other hand in asynchronous message passing, processes can stay active and carry out requests from others based on the order in their local queue, without waiting for their requests to be completed. In the presence of concurrent and asynchronous message exchanges as well as possible message loss or corruption, *consensus or state-machine replication protocols* are essential ingredients for maintaining strong consistency. Hence, developing practical implementations or reasoning about correctness of such protocols is notoriously difficult.

In this dissertation, due to differences stated perviously, we investigated concurrent data structures separately, based on their communication type between processes. Our main goal is to offer algorithmic approaches for improving reliability of these concurrent data structures. Mainly we focused on two research areas for the data structures that are built on top of shared memory: (1) checking safety by locating the errors (if there is any) and (2) repairing these errors. For the data structures that are built on top of a network, we analyzed particularly the consensus protocols and offered a new proof methodology for checking their safety according to the protocols' own specifications.

1.1 Checking Safety of Shared Memory Programs

1.1.1 Problem Statement

Our first objective is verifying concurrent data structures that are built on top of shared memory. We assume that the memory is sequentially consistent throughout this thesis. The de-facto correctness criterion for concurrent data structures is *linearizability*. It ensures that methods of these data structures behave as if they were executed *atomically*, one after the other to rely on the

(sequential) ADT specification. Violation of linearizability is witnessed by a *finite* erroneous execution in which the outputs of individual operations do not match those of a sequential execution of the same operations.

We are interested in the problem of checking linearizability of a given concurrent data structure. This problem can be seen as an instance of a more general problem which is checking a safety property for a given concurrent program. As mentioned above, linearizability violations are finite executions and therefore, linearizability is a safety property. The concurrent program is a client of the data structure, which calls methods of the data structure from some (possibly unbounded) number of threads. We will use the terms concurrent data structure and library interchangeably.

We investigate the problem of checking the correctness of a library in the context of a specific client by checking linearizability of its executions and locating the linearizability errors in it, if any.

1.1.2 The Approach

We will consider an approach which consists in applying model checking to a *bounded* client that consists of a bounded number of threads, each thread doing a bounded number of calls. Along with a complete enumeration of bounded clients, using a tool such as Violat [29], this approach is sound and complete in the limit (every linearizability violation can be captured in the context of a bounded client).

Model checking [24, 25, 84] explores the state space of a given program in a systematic manner and verifies that each reachable state satisfies a given property. It provides high coverage of program behavior, but it faces the infamous state explosion problem, i.e., the number of possible thread interleavings grows exponentially in the size of the source code.

To address the state explosion problem, we will use *partial order reduction* (POR) [24, 40, 41, 44, 51, 82, 93] which limits the number of explored interleavings without sacrificing coverage. Partial order reduction relies on an equivalence relation between interleavings, where two interleavings are equivalent if one can be obtained from the other by swapping consecutive independent (non-conflicting) execution steps. It guarantees that at least one interleaving from each equivalence class (called a Mazurkiewicz trace [74]) is explored. Optimal POR techniques explore exactly one interleaving from each equivalence class. Beyond this classic notion of optimality, POR techniques may aim for optimality by avoiding visiting states from which no optimal execution may pass.

1.1.3 State of the Art

There is a large body of work on POR techniques that address its soundness when checking a certain class of specifications for a certain class of programs, or its theoretical optimality (see Section 2.2.4 for more details). The set of interleavings explored by some POR technique is defined

by restricting the set of transitions that are explored from each state (scheduling point). We assume that individual threads are deterministic, and therefore, equate sets of transitions from a given state to a set of threads (that act in that state). Depending on the class specifications, assumptions about programs, or optimality targets, there are various definitions for this set of threads, including sleep sets[41], stubborn sets [41, 44, 93], persistent sets [40, 43, 60], ample sets [24, 51], and source sets [3].

The design of a model checking algorithm based on POR has to consider several computational tradeoffs. First, such an algorithm can be stateful or stateless [39], which corresponds to a tradeoff between memory consumption versus execution time. Stateful model checking records visited states, thereby consuming more memory, but stateless model checking performs redundant exploration from already visited states. Second, the computation of the set of threads that are explored from some state can be more or less complex. Focusing on theoretical optimality, i.e., exploring *exactly* one interleaving from each Mazurkiewicz trace, may make this computation more complex. This complexity in turn may diminish the overall performance when the potential for reducing the state space is not large, i.e., most Mazurkiewicz traces contain a small number of interleavings. In such a case, exploring more interleavings can take less time than computing more precise constraints on the explored schedules. Third, POR algorithms may compute the information they use for the purpose of reduction *statically*, by some kind of conservative static analysis of the source code, or *dynamically*, during the exploration of interleavings. Static computation is usually cheaper and less precise than dynamic computation.

1.1.4 Our Contributions

In this dissertation, we investigate the use of POR in model checking from a practical point of view. We propose several POR-based stateful model checking algorithms with a focus on overall performance rather than theoretical optimality. These algorithms combine static and dynamic computations of sets of threads to explore from a given state, whose definition is based on the recently proposed source sets.

We evaluated several possible variations of our algorithms and existing ones, implemented in the context of the Java Pathfinder (JPF) model checker [96]. We showed that the performance of the algorithms depends mainly on the potential for reduction in the state space. We recommended desirable usages of these algorithms based on the resources of the setup. We also analyzed the potential effects of different search strategies (enumerations of execution steps).

1.2 Root-Causing Linearizability Violations

1.2.1 Problem Statement

As a continuation of the work presented above, which focused on identifying linearizability violations, we consider the problem of *debugging* linearizability violations, finding their root-cause and suggest possible repairs. This is a difficult problem because linearizability violations can be large (even if they are obtained in the context of a client with few methods, the number of internal steps performed in those methods can be large) and interleave steps of different threads in complicated ways. Fixing a linearizability violation can be easy if one is willing to disregard or sacrifice performance, e.g., by enforcing coarse-grain atomic sections that span a whole method body. On the other hand, it is difficult to localize the problem to a degree that fixing it would not affect the otherwise correct behaviors of the concurrent data structure, and its performance.

We can tackle this problem in three steps. First of all, we should define some notion of *root-cause* for non-linearizability. Secondly, we need to come up with efficient algorithms to find the root causes of the bug. And as a third step, we must provide an effective methodology for removing the bug with that root cause.

In this dissertation, we consider the problem of identifying the root causes of linearizability errors by providing useful hints or guidelines to repair these errors when they are discovered. This is a problem relevant in practice because it is time consuming and very difficult for a developer to determine the origin of the violation manually and fix the program accordingly.

1.2.2 The Approach

The approach that we chose to identify the root causes of linearizability errors is finding code blocks (repairs) in program's code whose atomicity is required to rule out all given non-linearizable executions. The key insight of this approach is that the problem can be reduced to a simpler algorithmic problem of identifying minimal root causes of conflict serializability violations in an erroneous execution. In a concurrent program with a given decomposition of its code into code blocks, an execution is conflict serializable if it is equivalent to an execution in which all code blocks are executed in a sequential non-interleaved fashion. A repair can be retrieved using a decomposition of the set of operations in an execution into a set of blocks called intervals, such that this execution is not conflict serializable with respect to this decomposition. These intervals will correspond to atomic sections which will be recommended as repairs by using our approach.

1.2.3 State of the Art

There are various works such as [22, 54, 55, 56, 57, 59, 81, 99] for fault localization, error explanation, counterexample minimization and bug summarization for concurrent programs. These works try to derive a report for the reasons of encountered bug but our work differs from them due to the choice of correctness criterion which is linearizability.

Another branch of works [8, 15, 17, 23, 48, 73, 94, 95] for fixing concurrency errors is on synchronization synthesis techniques that can aid developers repair concurrency related bugs. Again, this group of works does not focus on linearizability as the correctness criterion, but rather on assertion local violations. Additionally, Afix [56] and ConcurrencySwapper [16] automatically repair bugs related to concurrency. Unlike our approach, ConcurrencySwapper works with error invariants to generalize a linear error trace to a partially ordered trace that is used for synthesizing a repair.

To our knowledge, Flint [70] is the only other approach that focuses on root-causing linearizability violations. However, it concentrates on only one particular abstract data type which is maps. Also differently than our approach, it is based on enumeration-based synthesis and does not depend on concrete linearizability errors.

1.2.4 Our Contributions

In this dissertation, we present a novel approach that diagnoses the root causes of linearizability errors. As explained above, this is done by producing repairs as code segments which must be non-interleaved with actions from other threads (executed sequentially) to reestablish linearizability.

We compute optimal repairs where optimality refers to allowing a maximal number of interleavings. Optimal repairs can be obtained by eliminating the ones that are subsumed by some other repair which disables the same set of non-linearizable executions. However, there can still be multiple optimal repairs and we need to determine which one is more likely to correspond to the root-cause. Therefore, we rank these optimal repairs with an heuristic by favoring the one that disable less linearizable executions. This heuristic relies on a hypothesis that cyclic memory accesses appearing in linearizable executions are not “dangerous”.

We propose theoretical results outlining the reduction of the main problem to one related to conflict serializability violations, and a polynomial algorithm to solve the latter. We have implemented our approach and carried out several experiments using realistic concurrent data structures to demonstrate its efficiency.

1.3 Verifying Safety of Consensus Protocols

1.3.1 Problem Statement

Implementing linearizable concurrent data structures that work over a network typically relies on algorithms used to solve a more generic problem called *state machine replication*. In state machine replication, the goal is to make a number of processes agree on a sequence of commands (transitions) on a state machine. Therefore, interpreting the state machine as an ADT specification, such algorithms can be used to implement a linearizable concurrent data structure.

In general, state machine replication protocols are part of a larger class of protocols called *consensus protocols*. Consensus protocols are constructed for agreement on a single value (*single-decree*) or multiple ordered values (*multi-decree*) which is the basis of state-machine replication protocols. A standard example of a single-decree consensus protocol is the classic Paxos [67] introduced by Lamport. Multi-decree consensus protocols can be defined as a composition of multiple independent instances of a single-decree protocol where each decided value (in some instance) will be matched with a unique sequence number to represent an order between these values. One such example is the Practical Byzantine Fault Tolerance (PBFT) protocol [14]. Multi-decree consensus protocols can also be defined as an agreement over a common log (e.g., Raft [102] protocol) or a branch in a tree (e.g., HotStuff [105] protocol).

In consensus protocols, processes receive requests (values) from clients and clients receive a confirmation response when processes agree on some (sequence of) values. Most of these protocols work in a succession of communication-closed rounds [27] and in each round, there is a unique designated leader (known by all processes) propagating the request of the client to all other processes. When the leader of some round becomes unresponsive, then the processes request to proceed to the next round to have a new leader. In each round, processes progress through several phases and in each phase, at least a *quorum* of responses are needed to advance to the next phase. During a phase, processes acknowledge received messages by their response if the messages satisfy some conditions (e.g., accepting at most one proposal for a certain round). If all the phases can be passed (by a quorum of processes) during some round, then the value that is proposed is decided in the same round and can be sent back to the client as a response.

In consensus protocols, some processes may stop arbitrarily (crash failure) or fail intentionally (Byzantine failure) and the system must be resilient to such failures, or in other words, *fault tolerant*. Byzantine failures occur when a process continues to operate but acts differently than its expected behavior, possibly due to malicious actions of an adversary. Adversarially acting faulty processes can send messages without the obligation of checking conditions specified in the protocol. Thus, Byzantine fault tolerant (BFT) protocols require additional cryptographic primitives to prevent an adversary from breaking the system by tricking correct processes with corrupted

messages. For instance, cryptographic collision-resistant hash functions [88] can be used in order to prevent an adversary to generate a message with the same hash but different content. Other fundamental primitives are related to public-key signatures [87] for preventing an adversary from acting on behalf of an honest (correct) process.

Developing correct practical implementations of consensus protocols or even reasoning about their correctness is very challenging. Their complexity stems from the different assumptions (i.e., process or network link failures, Byzantine failures etc.) on the environment they operate with as explained previously. The correctness of consensus protocols under these assumptions is expressed through the following properties:

- *Validity*: The decided value must be proposed by a process (who received it from a client).
- *Agreement*: Every correct process must decide on the same value.
- *Termination*: Eventually, there will be a decision on some value.

If a protocol satisfies the notions of *Validity* and *Agreement*, we say that the protocol is *safe*.

In this dissertation, we consider the problem of reasoning about the safety of consensus protocols. This is a very challenging problem and with many practical applications as mentioned above.

1.3.2 The Approach

Our approach in reasoning about the safety of consensus protocols is based on *refinement*. This consists in showing that the behaviors of a given protocol can be mapped to behaviors of an abstract operational specification (defined as a state machine) which is already proved to be safe (satisfies *Validity* and *Agreement*). This mapping must ensure that the safety of the specification implies the safety of the original protocol.

Designing useful abstractions can be very helpful in taming complexity while designing new protocols, or in gaining confidence about their correctness without going through ad-hoc and brittle proof arguments like in inductive invariant based safety proofs. Nevertheless, finding the right trade-off between the expressivity of an abstraction and the "easiness" of relating it to concrete protocols is an important and non-trivial issue in general.

1.3.3 State of the Art

The problem of proving the correctness of consensus or state-machine replication protocols has been studied in previous works. There are some works that use inductive invariants [78, 79], which however, are hard to invent since they have to describe all intermediate states produced by all possible orders of receiving messages. Some other works [6, 9, 36, 37, 38, 49, 63, 102] are based on

refinement. Most of these works [36, 37, 38, 49, 63, 102] use refinement to prove just one specific protocol and the abstractions that they derive during this process does not generalize beyond that protocol. A few of them such as [6, 9] studied more generic abstractions but relating a concrete protocol to such abstractions is quite cumbersome and involved (we give more details in Section 3.9). Intuitively, these abstractions are too permissive and make refinement proofs harder.

1.3.4 Our Contributions

We propose a novel abstract representation of the dynamics of consensus protocols which focuses on quorums of responses (votes) to a request (proposal) that form during a run of the protocol. We show that focusing on such quorums, a run of a protocol can be viewed as working over a tree structure where different branches represent different possible outcomes of the protocol, the goal being to stabilize on the choice of a fixed branch. This abstraction, formalized as a sequential tree object called QTree, resembles the description of recent protocols used in Blockchain infrastructures, e.g., the protocol supporting Bitcoin or Hotstuff. We show that this abstraction supports reasoning about the safety of various algorithms, e.g., Paxos, PBFT, Raft, and HotStuff, in a uniform way. In general, it provides a novel induction based argument for proving that such protocols are safe.

Proving that a given protocol refines QTree is similar to proving that a concurrent data structure admits *fixed linearization points*, i.e., each method takes effect whenever executing a fixed statement in its code. That is, one has to identify specific steps of the protocol that correspond to executing some QTree method.

We show that QTree supports reasoning about the safety of consensus protocols in a *uniform* way. In general, it provides a novel induction based argument for proving that such protocols are safe. Compared to previous works, we demonstrate applicability to a much larger class of protocols: state machine replication protocols, single-decree and multi-decree consensus protocols with various optimizations. We believe that QTree is also helpful in taming complexity while designing new protocols, or in gaining confidence about their correctness without going through ad-hoc and brittle proof arguments.

1.4 Thesis Outline

The rest of this dissertation is organized as follows:

- Chapter 2 focuses on finding and fixing linearizability violations in concurrent data structure implementations:

  - After the preliminary material in Section 2.1, Section 2.2 introduces new model checking algorithms based on POR that focus on improving overall performance.
  - Section 2.3 describes an approach for identifying root causes of linearizability violations and suggesting possible repairs.

- Chapter 3 presents the QTree abstraction of consensus protocols and shows how to establish refinement for a diverse class of protocols.
- Chapter 4 concludes the dissertation with final remarks and future research directions.

2 Debugging Linearizability in Shared-Memory Concurrent Data Structures

In this chapter, we present new methodologies for discovering and repairing linearizability bugs in concurrent data structures implemented on top of shared memory. Section 2.1 introduces a set of definitions that will be used throughout this chapter about multi-threaded programs, linearizability, and partial order reduction techniques. Then in Section 2.2, we investigate several techniques for improving the performance of POR-based model checking algorithms. Section 2.3 introduces a method for root-causing and repairing linearizability bugs by identifying minimal code segments that must be executed in isolation (without interference from other threads).

2.1 Preliminaries

Assuming that programs run under sequential consistency, we model a multi-threaded program implemented on top of a shared memory with a bounded number of threads as a labeled transition system (LTS) $L = (\mathcal{S}, s_I, \Gamma, \mathcal{A})$, where

- $\mathcal{S}$ is a (possibly infinite) set of states s representing a set of *shared* objects visible to all threads and a finite set of *local* objects visible to a single fixed thread, and a program counter for each thread.
- The state $s_I \in \mathcal{S}$ is the unique initial state.
- $\mathcal{A}$ is a set of actions a where a is an *action* (transition label) representing the execution of an *atomic* statement in the code.
- Γ is a set of labeled transitions (s, a, s') such that $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$.

There are two main types of actions:

- *invisible actions*: $a = (t, k, pc, type, \epsilon)$ where a thread t executes a statement of a method invocation $k \in \mathbb{N}$ (we assume method invocation identifiers to be natural numbers) at

program counter pc that accesses no shared object. For simplicity, we consider only two specific $types$ of invisible actions related to calling or returning from a method, which will be formalized shortly.

- *visible actions*: $a = (t, k, pc, r/w, o)$ where t executes a statement of a method invocation k at pc and its $type$ is either r or w which represent reading from and writing to a shared object o, respectively.

We fix an arbitrary set $\mathbb{T}$ of thread ids. For an action a, $tid(a)$ is the thread id $t \in \mathbb{T}$, $act(a)$ is the action $type$ and $obj(a)$ is the accessed shared object when a is visible (otherwise it is undefined). The method invocation identifier of an action a is denoted by $iid(a)$.

A transition labeled by a visible (resp. invisible) action is called visible (resp. invisible). In the context of a full-fledged programming language, invisible transitions are related to control-flow manipulations (e.g., calling or returning from a method, starting and joining threads), local computations or accesses to "low-level" shared objects that are irrelevant for the intended (functional) specification. Visible transitions correspond to the execution of a single access to a shared object followed by a maximal sequence of accesses to the local objects.

We fix arbitrary sets $\mathbb{M}$ and $\mathbb{V}$ of method names and parameter/return values. For given sets $\mathbb{M}$ and $\mathbb{V}$ of method names and values, we formalize invisible actions related to calling (a_{call}) or returning from (a_{ret}) a method. For each call action a_{call} of a method invocation k, $type$ is $call\ m(v)$ such that $m \in \mathbb{M}$ and $v \in \mathbb{V}$. Similarly, for each return action a_{ret} of a method invocation k $type$ is $ret\ v$ such that $v \in \mathbb{V}$. Method invocation identifiers are used to pair call and return actions.

An action a is *enabled* in state s if there exists s' such that $(s, a, s') \in \Gamma$. We use $next(s, t)$ to denote the transition $(s, a, s') \in \Gamma$ for some a and s' with $tid(a) = t$, if it exists, and $succ(s, t)$ to denote the successor s' in this transition. Otherwise, we say that t is *blocked* in s. The set $enabled(s)$ is the set of threads that are not blocked in s. A state s is *final* if $enabled(s) = \emptyset$.

Two actions a and a' of different threads are *independent* if they are both enabled in a state s and either one of them is an invisible action, or they are both visible and access different shared objects ($obj(a) \neq obj(a')$), or they both perform a read access ($act(a) = act(a') = r$). The actions a and a' are called *dependent*, denoted by $a \nsim a'$, if they are not independent. We assume that if an action a enables or disables another action a', then $a \nsim a'$. Two transitions are (in)dependent if and only if they contain actions that are (in)dependent.

An *execution* E from a state s is a finite sequence of alternating states and actions such that $E = s_0, a_0, s_1, a_1, \ldots, s_n$ with $s_0 = s$ and $(s_i, a_i, s_{i+1}) \in \Gamma$ for each $0 \leq i \leq n - 1$. We assume all executions satisfy standard well-formedness properties, e.g., in the projection of E on considered actions (above) of the same thread, every return action is preceded by a call action of

the same method invocation k where all the other actions between them are also the actions of k. We also assume that every atomic section (block) is interpreted as an *uninterrupted* sequence of actions that correspond to the instructions in that atomic section.

The set of executions starting from s in the LTS L is denoted by $E(L, s)$. An *initialized* execution is an execution from s_I. Initialized executions that end with a final state are called *full* executions.

The *happens-before* relation in an execution E, denoted by $\rightarrow_E$, captures the causal relation among actions in E, where $a_1 <_E a_2$ represents that a_1 occurs before a_2 in E (we omit the subscript E and use just $a_1 < a_2$ if E is clear from the context), and $\rightarrow_E = \mathsf{po}_E \cup \mathsf{cf}_E$ where

- the *program order* relation po_E relates any two actions a_1 and a_2 with the same thread id such that $a_1 <_E a_2$, and
- the *conflict* relation cf_E relates any two actions a_1 and a_2 with different threads ids that accessing the same shared object, at least one of them being a write, such that $a_1 <_E a_2$.

Given two actions a and a' in E, $a \rightarrow_E a'$ holds iff $a \nsim a'$ and $a < a'$. Two executions E and E' are called *equivalent* if $\rightarrow_E = \rightarrow_{E'}$ ($\mathsf{po}_E = \mathsf{po}_{E'}$ and $\mathsf{cf}_E = \mathsf{cf}_{E'}$) and they are called cf*-equivalent* when only $\mathsf{cf}_E = \mathsf{cf}_{E'}$. For a full execution E, we use $[E]$ to denote the set of full executions E' that are equivalent to E.

Given an LTS $L = (\mathcal{S}, s_I, \Gamma)$ that models a concurrent program, an LTS $L_r = (\mathcal{S}_r, s_I, \Gamma_r)$ with $\mathcal{S}_r \subseteq \mathcal{S}$ and $\Gamma_r \subseteq \Gamma$ is called *sound for* L if for each full execution E of L, there exists a full execution E' of L_r that is equivalent to E.

2.1.1 Linearizability

We model a *library* $\mathcal{L}$ containing a set of methods $\mathbb{M}$ with an LTS L which models all the executions of L under a most general client that makes an arbitrary number of invocations of methods in $\mathbb{M}$ from a arbitrary number of threads. The projection of an execution E of a library $\mathcal{L}$ over call and return actions of methods in $\mathbb{M}$ is called a *history* and denoted by $h(E)$. A history is *sequential* when each call action a_{call} is immediately followed by a return action a_{ret} of the same method invocation $k = iid(a_{ret}) = iid(a_{call})$.

A *linearization* of a history h_1 is a sequential history h_2 that is a permutation of h_1 which preserves the order between return and call actions, i.e., a given return action occurs before a given call action in h_1 if and only if the same holds in h_2. An execution E of a library $\mathcal{L}$ is *linearizable* if $\mathcal{L}$ contains some sequential execution whose history is a linearization of $h(E)$. Intuitively, E is linearizable if for every call action a_{call} followed by a return action a_{ret} with $iid(a_{call}) = iid(a_{ret})$, there exist a *linearization point* at some moment between the occurrences of a_{call} and a_{ret} where

the invocation takes effect instantaneously. A library is *linearizable* if all its executions are linearizable[1]. In the following, since *linearizability* [50] is used as the main correctness criterion, a *bug* of a library is an execution E that is not linearizable.

2.1.2 Partial Order Reduction

The purpose of POR is to reduce the number of states to be searched by model checking without sacrificing coverage, i.e., ensuring that at least one element (exactly one if it is optimal) from each equivalence class called Mazurkiewicz trace is discovered. The set of executions explored by POR techniques is defined by restricting the set of threads whose actions are explored from each state. Needed information to restrict the threads can be gathered either statically by observing the source code a-priori or dynamically by analyzing the already explored state space on-the-fly, during the search of other states. There are already various definitions for restricted sets of threads such as stubborn sets, persistent sets, source sets etc. where all guarantee soundness, i.e., at least one execution from each equivalence class is explored. Note that, we define persistent and source sets as sets of threads, which correspond to sets of transitions in the classical sense, under the assumption of determinacy of individual threads.

The algorithms discussed in this chapter fall into two categories based on their restricting set of threads definition: persistent sets and source sets. We selected these two sets because all of the algorithms before the introduction of source sets that are based on other sets such as ample sets, stubborn sets etc. are actually computing persistent sets and hence, persistent sets generalize others [40]. On the other hand, source sets are introduced recently in 2017 by Abdulla and it is an advancement of the technique based on persistent sets since they are provably optimal [2] (i.e., the set of explored threads from some state must be a source set in order to guarantee exploration of all Mazurkiewicz traces).

Intuitively, a set of threads T is *persistent* for a state s if in any execution starting from s, the first transition that is dependent on some transition starting from s of some thread $t \in T$ is taken by some thread $t' \in T$ (t and t' may be equal). A set of threads T is a *source* set for s if for any execution starting from s, there is some thread in T that can take the first step, modulo reorderings of independent transitions.

Definition 1 (Persistent Set [40])**.** *A set of threads T is called a* persistent set for a state s *if for every execution E from s that contains only transitions from thread $t' \notin T$, every transition in E is independent of every transition $next(s, t)$ with $t \in T$.*

[1] Linearizability is typically defined with respect to a sequential ADT. Here, we take the simplifying assumption that the ADT is defined by the set of sequential histories of the library. This holds for all concurrent libraries that we are aware of.

For an execution E from a state s that ends in a final state, a thread t is called a *weak initial* of E if there exists an execution E' that is equivalent to E and starts with a transition of t.

Definition 2 (Source Set [3])**.** *A set of threads T is called a* source set for a state s *if every execution from s that ends with a final state has a weak initial thread in T.*

An exploration where each state is expanded w.r.t. the threads in a persistent or source set is sound [2] (when finished under the assumption of absence of deadlocks, i.e., a full execution E contains every action enabled in a state of E, it produces an LTS that is sound for the "full" LTS of the program which is deterministic and acyclic). However, source sets guarantee a stronger notion of optimality as mentioned previously. There exist programs where any persistent set (for the initial state) is strictly larger than a source set, but every persistent set is also a source set [2]. Note that source sets are monotonic in the sense that any superset of a source set is also a source set. This property does not hold for other definitions such as stubborn sets, persistent sets, or ample sets.

To demonstrate the optimality of source sets, we go through the example from [3]:

$1:$	$2:$	$3:$
$w\ x \quad (a)$	$r\ y$	$r\ z$
	$r\ x \quad (b)$	$r\ x \quad (c)$

Figure 2.1: Example code taken from [3]

In Figure 2.1, flow of a simple program with three threads (represented with 1, 2 and 3) is depicted. There exists a shared object x where each thread contains a single transition that accesses it and these transitions are denoted with (a), (b) and (c), respectively. Since other transitions are independent from the rest, only the permutations of (a), (b) and (c) can lead to distinct Mazurkiewicz traces. As both (b) and (c) are reading from x, we have six different orders between these transitions that lead to four different equivalence classes in total:

1. (a), (b), (c) or (a), (c), (b)
2. (b), (a), (c)
3. (c), (a), (b)
4. (b), (c), (a) or (c), (b), (a)

In all the POR algorithms, when a new state is visited for the first time, one of its enabled transitions from a certain thread must be selected to proceed and since it is a new state, there is no

information from the future of this transition that can be used for the selection. For instance in the initial state s of the state space that is generated from executing the code in the given example, all three threads are enabled and one can select the first transition to be executed based on the ascending order of thread ids of these transitions. In this case (a) will be the first transition, and as (a) is dependent on both future transitions (b) and (c), {1, 2, 3} is the only persistent set of s that can contain 1. But source set based algorithms permit {1, 2} or {1, 3}, as both 2 and 3 are weak initial threads. Thus, {1, 2} and {1, 3} are the only two optimal sets that can contain 1 whereas {1, 2, 3} is clearly not optimal. Hence, a source set based algorithm can explore exactly one trace from each of those 4 equivalence classes even the first transition to be executed is selected from thread 1, but this is not true for persistent set based algorithms as they must explore at least both (b), (c), (a) and (c), (b), (a) orders (even though exploring only one of them is sufficient) because of the persistent set definition.

2.2 A Pragmatic Approach to Stateful Partial Order Reduction

In this Section, we investigate the use of partial order reduction (POR) from a practical point of view. In the context of verifying concurrent data structures, we investigate the following research question: what tradeoffs in POR families of algorithms may lead to practical net gains in verification or bug-finding times? We focus on the application domain of verification of Java concurrent data structures using a tool like Violat [29] which can be utilized to collect client programs that admit executions leading to linearizability bugs, using assertions to check that any combination of return values observed in an execution belongs to a statically precomputed set of ADT-admitted return-value outcomes. Violat is integrated with the Java Pathfinder (JPF) model checker [96], which enables complete systematic coverage of a given test program and outputting executions leading to consistency violations, thus facilitating diagnosis and repair. We investigate POR algorithms implementable in JPF.

We study several stateful model checking algorithms with POR in the context of Violat's test programs. This choice was inspired by experiments that demonstrated that it is much faster than the stateless variation (see Section 2.2.3). We introduce POR algorithms that combine static and dynamic computations of sets of threads to explore from a given state. In the context of stateful model checking, static techniques may seem like the better option. A dynamic computation usually requires re-traversing the state space starting in an already visited state which can be time consuming. Note however that re-traversing the state space that is already loaded in memory takes less time than generating that state space in the first place, which involves executing program statements.

Our starting point is a simple static POR algorithm, called S-POR, that makes use of invisible transitions which correspond to the safe actions introduced in [51]. Based on a syntactic analysis of the code, we identify shared and synchronization objects, and assume that every transition that does not access such an object is invisible. For clients of concurrent data structures, such objects correspond to class fields accessed in a method of the data structure. The S-POR algorithm prioritizes the exploration of invisible transitions over visible ones, i.e., if an invisible transition is enabled in a given state then this is the only explored transition from that state, and otherwise, all enabled transitions are explored. We demonstrate that S-POR has a small overhead and the potential for substantial reductions, and therefore leads to significant speedups with respect to standard JPF which employs a very conservative heuristic for its POR (see Section 2.2.3).

S-POR is effective, but by the nature of being lightweight, does not always reduce the state space effectively. We introduce two new algorithms as extensions of S-POR, with the idea of performing a more aggressive reduction while keeping the overhead reasonably low. They *dynamically*

compute source sets, which restrict the set of threads explored from a state with only visible enabled transitions. We focus on source sets since they are provably minimal and monotonicity of source sets makes their computation less sensitive to the order in which transitions from a given state are enumerated, compared to definitions such as persistent set.

The design principle behind our algorithms is to favor efficiency over theoretical optimality. Our algorithms are not theoretically optimal. However, we demonstrate that they are more efficient than the optimal algorithm [3] where the overhead of source set computation subsumes any gains from not exploring the redundant interleavings.

In general, a dynamic computation of source sets relies on tracking dependencies between actions in the already explored executions. Our two proposed algorithms differ in the way in which the tracking is performed: one is *eager* and called DE-S-POR, and the other is *lazy* and called DL-S-POR. Intuitively, DE-S-POR advances the computation of source sets for predecessors in the current execution in a style similar to previous dynamic POR algorithms, e.g. [3, 35], while DL-S-POR advances the computation of the source set in a given state only when the exploration backtracks to that state and one must decide if a new transition has to be explored.

The thesis of this section is that when there is a big enough potential for reducing the state space of a concurrent program, i.e., many Mazurkiewicz traces are large enough, non-optimal but carefully customized algorithms, like DE-S-POR and DL-S-POR, can have the largest impact compared to the two extremes of the spectrum, that is, S-POR or theoretically optimal algorithms like [3]. If the potential for reduction is small, then a simple static algorithm like S-POR provides the best overhead-gain tradeoff.

To support this thesis, we implemented these algorithms in JPF and evaluated them on a number of clients of concurrent data structures from the Synchrobench repository [46]. Our evaluation shows that they outperform (1) their variations that are directly built on top of the standard setup of JPF, (2) their stateless variations, and (3) a best-effort implementation of a stateful variation of the optimal algorithm in [3]. The lazy algorithm DL-S-POR is more efficient than the eager DE-S-POR, and more efficient than S-POR on clients with a big enough potential for reducing the state space.

The remainder of this section is organized as follows:

- In Section 2.2.1, we present two stateful POR algorithms; first S-POR that has no dynamic computation and then DE-S-POR (which is build on top of S-POR) that computes source sets on-the-fly while searching the state space.

- In Section 2.2.2, we introduce another stateful POR algorithm DL-S-POR (which is again build on top of S-POR) that computes source sets lazily by need (when it backtracks to a state which has still undiscovered enabled transitions) using additional traversals of the

explored state space. Then we talk about some optimizations that can be done on DL-S-POR but not possible on DE-S-POR.

- In Section 2.2.3, we present the empirical evaluation of our algorithms by comparing them based on time and space consumption and show how different enumerations of transitions can affect the bug finding times.
- Section 2.2.4 presents related work.

2.2.1 Eager Source Set POR (DE-S-POR)

We present a first stateful POR algorithm that selects a sufficient set of threads to expand a state based on two criteria: (1) a static criterion based on (in)visible actions, and (2) a dynamic criterion based on *source sets* computed on-the-fly during the exploration. Source sets are maintained *eagerly* for each new transition that is explored, in a style similar to previous algorithms [3, 35]. For presentation reasons, we start with a simplified version that includes only the static criterion and continue with the full version afterwards.

2.2.1.1 Safe Set POR (S-POR)

Algorithm 1: Safe Set POR (S-POR)

Initialize: $Stack \leftarrow \emptyset$; $Stack.\mathbf{push}(s_I)$; $L_r \leftarrow \emptyset$;

```
1 Explore()
2     s ← Stack.top;
3     if notVisited(s) then
4         forall t ∈ safeSet(s) do
5             (s, a, s') ← next(s, t);
6             Stack.push(s'); // (s, a, s') is added to L_r
7             Explore();
8             Stack.pop();
```

$notVisited$(s) holds if s is final in L_r but $enabled$(s) $\neq \emptyset$

$$safeSet(s) = \begin{cases} \{t\}, & \exists t \in enabled(s) : next(s,t) = (s,a,s') \text{ and } a \text{ is invisible} \\ enabled(s), & \text{otherwise} \end{cases}$$

Algorithm 1 presents a stateful DFS traversal of a concurrent program, represented by an LTS, which restricts the traversal to so called *safe sets*. Figure 2.2 illustrates the core idea of this algorithm. The safe sets prioritize the exploration of *invisible* transitions over visible ones.

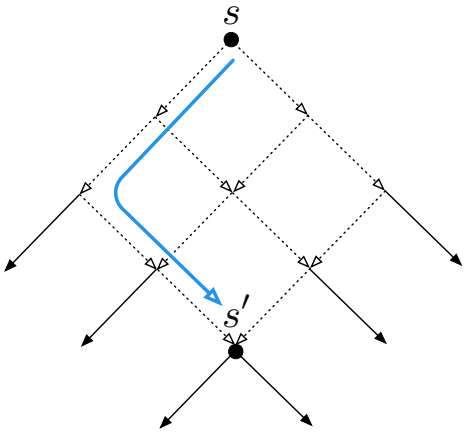

Figure 2.2: Full traversal vs. partial safe set POR (in blue)

For a state s, if there is an enabled thread $t \in enabled(s)$ whose enabled transition is invisible, then $safeSet(s) = \{t\}$. Otherwise, $safeSet(s)$ contains all the threads enabled in s, and s is called an *irreducible state*. In Figure 2.2, only state s' is irreducible since any other state has at least one enabled invisible transition, and all other states are reducible.

In Algorithm 1, $Stack$ represents the stack of the DFS traversal and it is considered to be a global variable, and L_r records transitions explored during the traversal. Note that the DFS traversal stops the exploration whenever it visits a state s that has been visited in the past (see the condition at line 3). The choice of safe sets then provides additional savings on top of the standard DFS traversal strategy. When the traversal ends, L_r is sound (for the "full" LTS of the program).

Observe that Algorithm 1 can reduce the number of visited states in a significant way. The diagram in Figure 2.2 corresponds to a *fully explored* program LTS while the path marked by the blue arrow is the result of Algorithm 1. It is easy to observe that one can obtain an exponential reduction (with the base of the number of consecutive invisible transitions and the exponent of the number of threads) with this algorithm.

2.2.1.2 Full Algorithm

Algorithm 2 builds on top of Algorithm 1 by computing on-the-fly source sets to limit exploration of transitions from the *irreducible* states. More precisely, reducible states are traversed according to the strategy of Algorithm 1 (i.e., only one enabled invisible transition is followed) and for irreducible states, source sets determine what transitions are followed. Since safe sets are also source sets, the overall algorithm remains sound if the new source sets are computed correctly.

Figure 2.3 provides a declarative description of the key components of Algorithm 2. For a state s in the current execution (stored on the stack), the $s.\mathbf{current}$ set may be updated every

Algorithm 2: EAGER SOURCE SET POR (DE-S-POR)

```
   Initialize: Stack ← ∅; Stack.push(s_I); L_r ← ∅;
1  Explore()
2      s ← Stack.top;
3      if notVisited(s) then
4          if ∃t ∈ safeSet(s) then
5              s.backtrack ← {t};
6              s.current ← ∅;
7              s.done ← ∅;
8              while ∃t' ∈ s.backtrack \ s.done do
9                  (s, a, s') = next(s, t');
10                 Stack.push(s');
11                 s.done = s.done ∪ {t'};
12                 s.current[t'] ← {t'};
13                 if a is visible then
14                     UpdateCurr(a)
15                 Explore();
16                 s.backtrack ← UpdateBack(s, a);
17                 Stack.pop();
18     else
19         A_s ← {a' : a' occurs in an execution from E(L_r, s)};
20         foreach a' ∈ A_s do UpdateCurr(a');
21 UpdateCurr(a)
22     E is the initialized execution of L_r following states in Stack;
23     (s, a', s') is the last transition of E with a ≁ a' ∧ tid(a) ≠ tid(a')
24     if (s, a', s') ≠ null then
25         s.current[tid(a')] = s.current[tid(a')] ∪ {tid(a)};
```

time a new visible transition is explored, and the $s.\mathbf{backtrack}$ set may be updated every time the exploration backtracks to s. The update of $s.\mathbf{backtrack}$ relies on the sets $s.\mathbf{current}$ computed while traversing successors of s.

When a new transition (s, a, s') from a state s is traversed, the active thread $tid(a)$ is added to the current set $s_l.\mathbf{current}[t]$, where s_l is the last state from which the current execution performs a transition that is dependent on a such that $t \neq tid(a)$ is the thread of that transition. See line 14 and the $\mathbf{UpdateCurr}$ function. When a transition is followed to a visited state s, the same update is done for *every* transition that is reachable from s, as if these transitions are traversed again. See lines 19-20 and note that the declarative definition of A_s at line 19 corresponds to a traversal of all the executions starting from s. This may be time-consuming, and yet, such updates

$$UpdateBack(s,a) = \begin{cases} safeSet(s), & \exists t \in s.\mathbf{current}[tid(a)] \setminus safeSet(s) \\ s.\mathbf{done}, & \exists T \subset s.\mathbf{done} : T = \bigcup_{t \in T} s.\mathbf{current}[t] \\ \bigcup_{t \in s.\mathbf{done}} s.\mathbf{current}[t], & \textbf{otherwise} \end{cases}$$

$s.\mathbf{current}[t]$: set of threads that execute a transition dependent on $next(s,t)$ which appears after it in an execution.
$s.\mathbf{done}$: set of threads whose transitions have been fully explored from s.
$s.\mathbf{backtrack}$: when equal to $s.\mathbf{done}$, a source set for s.

Figure 2.3: Description of important components in Algorithm 2.

are unavoidable in stateful POR algorithms because the current execution reaching s (stored on the stack) may belong to a different Mazurkiewicz trace compared to a previous execution reaching s (whose sequence of transitions leading to s was different).

When backtracking to a state s, the set $s.\mathbf{backtrack}$ is updated to take into account the transitions which are dependent on and occur after the last explored transition starting from s, called τ_s. If τ_s is a transition of thread t, the threads performing those dependent transitions are stored in $s.\mathbf{current}[t]$. If there is a dependent transition τ performed by a thread t' that is not in the safe set of s, then $s.\mathbf{backtrack}$ is updated conservatively to contain the safe set of s. This situation occurs when τ becomes enabled after executing some other thread t'' enabled in s, and observing an execution where τ_s occurs after τ requires first executing the transition of t''. Otherwise, the algorithm checks to see if a subset T of threads enabled in s which have already been explored are sufficient to cover $s.\mathbf{current}[t]$, and that T's transitions in s are independent from future transitions of threads not in T. In that case, $s.\mathbf{backtrack}$ is assigned with $s.\mathbf{done}$ and the exploration from s is halted. The subset of threads T defines a persistent set **and** a source set for s. Since source sets are monotonic, $s.\mathbf{done}$ is a source set for s. If none of the previous conditions hold, then $s.\mathbf{current}[t]$ is simply added to $s.\mathbf{backtrack}$. This computation is defined by the macro $UpdateBack$ in Fig. 2.3.

We illustrate the algorithm using Figure 2.4. In (a), s is reached for the first time and the transition labeled by $(1, w, o_1)$ is selected first to be executed. This is a visible transition of thread 1 that writes to the shared object o_1. After this transition is taken, $s.\mathbf{current}[1]$ and $s.\mathbf{done}$ become $\{1\}$. In (b), from some state which is reached later, the transition labeled by action $(2, r, o_1)$ is selected to be followed next. Since this action is dependent on $(1, w, o_1)$, thread 2 is added to $s.\mathbf{current}[1]$. Then in (c), a transition of thread 3 with an action dependent on $(1, w, o_1)$ is taken, and 3 is added to $s.\mathbf{current}[1]$. After backtracking to s in (d), $s.\mathbf{backtrack}$ is updated by simply copying $s.\mathbf{current}[1]$. The next transition to be taken from s belongs to thread 2

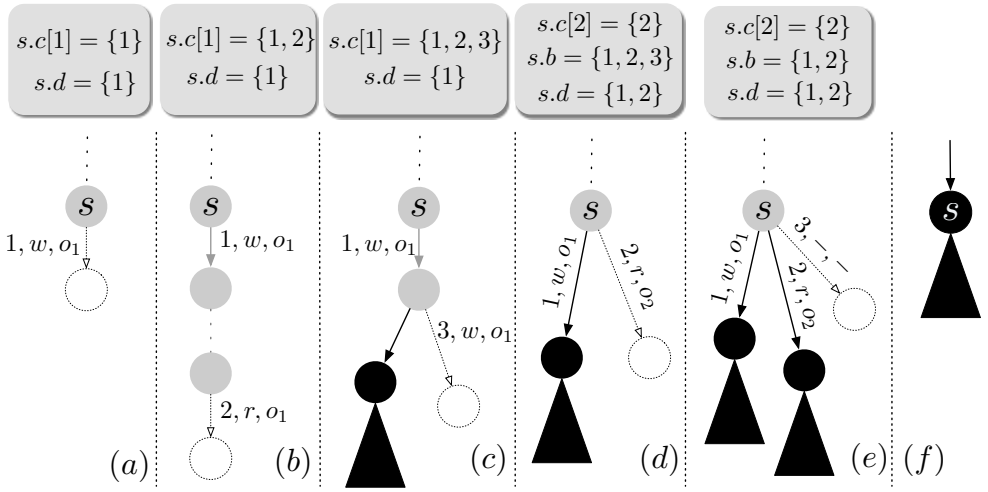

Figure 2.4: An example for Algorithm 2. Solid grey circles represent states stored on the stack, hollow dotted circles represent states on the top of the stack, and solid black circles represent states from which the exploration has been completed, i.e. their **backtrack** sets are equal to their **done** sets. Transitions follow the same pattern: dotted transitions are the latest to have been explored, solid grey ones are between states on the stack, and solid black ones are the ones taken in the past. Solid black triangles represent completed explorations starting from some state. We omit program counters from actions. **backtrack**, **current**, and **done** are abbreviated by the first letter.

which is in $s.\mathbf{backtrack} \setminus s.\mathbf{done}$. This entails the initialization of $s.\mathbf{current}[2] = \{2\}$ and the addition of 2 to $s.\mathbf{done}$. In (e), we backtrack to s for the second time and without having changed $s.\mathbf{current}[2]$. This means that $(2, r, o_2)$ is independent of any later action of another thread. Therefore, $\{2\}$ is a persistent set of s and $s.\mathbf{done} = \{1, 2\}$ a source set of s, and $s.\mathbf{backtrack}$ is assigned with $s.\mathbf{done}$ to stop the exploration from s, as pictured in (f).

This example shows that Algorithm 2 explores sets of transitions from a given state s that may correspond to a source set which is *not* a persistent set. The exploration in Figure 2.4 stops when $s.\mathbf{backtrack} = s.\mathbf{done} = \{1, 2\}$, but the only persistent set that includes thread 1 is $\{1, 2, 3\}$.

2.2.1.3 Soundness

Theorem 1. *Given a program represented by an LTS L, Algorithm 2 terminates with an LTS L_r that is sound for L.*

Proof. Based on the soundness of source sets (see Section 2.1.2), it is enough to show that for every state s in L_r,

$$s.\mathbf{backtrack} \text{ is a source set for } s \text{ in } L \text{ when it becomes equal to } s.\mathbf{done} \tag{2.1}$$

Due to the condition of **while** in line 8 of Algorithm 2, equality of $s.\mathbf{backtrack}$ and $s.\mathbf{done}$ is the only condition for stopping an exploration from a state s. Therefore, if some successor state s' of s is already explored and the search is backtracked to s, then $s'.\mathbf{backtrack} = s'.\mathbf{done}$, because otherwise, **while** loop in line 8 wouldn't be terminated for s'. Since $s.\mathbf{done}$ keep tracks of threads whose enabled transitions from s is already executed, the proof is reduced to showing that the following proposition holds:

$$\text{For any state } s,\ s.\mathbf{backtrack} \text{ is a source set when the exploration from } s \text{ is finished} \tag{2.2}$$

If s is a reducible state, then only one transition is explored from s which is an invisible transition. The fact that the thread performing this invisible transition is a persistent set and hence a source set follows directly from definitions as every persistent set is also a source set. When $s.\mathbf{backtrack} = s.\mathbf{done}$ is different from $safeSet(s) = enabled(s)$ (which is trivially a source set), it must be the case that there exists $T \subset s.\mathbf{done}$ such that $T = \bigcup_{t \in T} s.\mathbf{current}[t]$ due to the definition of $UpdateBack$ method. Now we show that T is a persistent set for s in L. Assume by contradiction that this is not the case, then due to the definition of persistent set, L admits an execution E starting from s that contains only transitions of threads different from those in T and at least one of these transitions τ of a thread $t' \notin T$ is dependent on some transition $next(s, t)$ with $t \in T$. For every $t \in T$, the successor state s' of s reached by $next(s, t)$ must be in L_r. Due to deadlock freedom assumption, some transition that has the same transition label with τ must be enabled eventually in some successor state of s'. Let $E' \in L_r$ be that execution from s which starts with $next(s, t)$ and contains such transition that shares the same label with τ. Now we will move forward by showing that the following proposition is correct, which will be used in the rest of the proof:

$$\text{If } L_r \text{ admits an execution } E'' \text{ from } s \text{ whose last transition that depends on and occurs before } \tau' \text{ is } next(s, t) \text{ where } act(\tau') = act(\tau) \text{ and } tid(\tau') = tid(\tau) = t', \text{ then } t' \in T \tag{2.3}$$

When τ' is executed from some successor state of s, t' will be added to $s.\mathbf{current}[t]$ (and eventually to T due to $UpdateBack$ method) by invoking the **UpdateCurr** function as $next(s,t)$ will be the transition in line 23 of Algorithm 2. This contradicts the assumption of $t' \notin T$ and therefore, it is enough to show that proposition below is correct for concluding the proof:

$$\text{If } L_r \text{ admits such an execution } E', \text{ then } L_r \text{ admits such an execution } E'' \tag{2.4}$$

To show that Proposition 2.4 holds, we proceed by induction on the order of $next(s,t)$ when we go backwards in E'. The base step is trivial since E'' can be E' when $next(s,t)$ is the first transition. Assuming by induction that $next(s,t)$ is the n-th transition that is dependent on and occurs before τ' in E' and L_r admits such execution E'', we show that this also holds when $next(s,t)$ is (n+1)-th transition with the same properties. Let s'' be the state that is reached from s by executing E'_p which is the prefix of E' until (not included) the last transition τ'' that is dependent on and occurs before τ' and hence, τ'' is enabled in s''. Using Proposition 2.3, as $tid(\tau')$ must be in T of s'', there must be another execution E''' from s'' such that τ''' occurs before τ'' where $act(\tau') = act(\tau''')$ and $tid(\tau') = tid(\tau''') = t'$. Since in the execution starting from s as E'_p and continues as E''', $next(s,t)$ is the n-th transition that is dependent on and occurs before τ''' (when we go backwards), proposition 2.4 is correct by using induction assumption. As mentioned, this contradicts the assumption of $t' \notin T$ in proof of proposition 2.2 and thus, T is a persistent set and a source set. By monotonicity of source sets, $s.\mathbf{backtrack}$ is also a source set. □

2.2.2 Lazy Source Set POR (DL-S-POR)

Algorithm 2 tracks dependencies between transitions in an eager manner, i.e., every new transition leads to updates of **current** sets. In this section, we present a lazy variation that computes such dependencies only when the exploration backtracks to a state. The incentive is to compute such dependencies only when needed to decide if the exploration from a given state should continue or not. Also, this enables several optimizations when traversing the state space to compute such dependencies that are not possible in the eager version.

Algorithm 3 presents our POR algorithm based on a lazy computation of source sets. Rather than updating the **current** sets on-the-fly for states on the stack, this algorithm re-traverses part of the state space each time it backtracks to a state s in order to update just **current** sets of s. This is done in the function **IsComplete**. As a result, $s.\mathbf{done}$ is populated with a new thread t just before computing dependencies with t's transition in s and not after executing that transition in the style of Algorithm 2 (see line 19). For every transition τ of a thread t from s

Algorithm 3: LAZY SOURCE SET POR (DL-S-POR)

```
   Initialize: Stack ← ∅; Stack.push(s_I); L_r ← ∅;
1  Explore()
2      s ← Stack.top;
3      s.backtrack ← ∅; s.done ← ∅; s.current ← ∅;
4      while true do
5          if ∃t_1 ∈ s.backtrack \ s.done then
6              t ← t_1
7          else
8              choose t ∈ safeSet(s) \ s.done
9          (s, a, s') = next(s, t);
10         Stack.push(s');
11         if notVisited(s') then
12             Explore();
13         if IsComplete(s) then
14             Stack.pop();
15             return
16         Stack.pop();
17 IsComplete(s)
18     forall (s, a, s') ∈ {s' ∈ L_r : t = tid(a) ∉ s.done} do
19         s.done = s.done ∪ {t};
20         T ← safeSet(s);
21         if s.done = T ∨ (∀t' ∈ T : isVisited(succ(s, t')) ∨ t' ∈ s.done) then
22             add transitions (s, a, s') to L_r;
23             s.done ← T;
24             s.backtrack ← s.done;
25             return true;
26         s.current[t] ← {t};
27         A_{s'} ← {a' : a' occurs in an execution from E(L_r, s')};
28         s.current[t] = s.current[t] ∪ {tid(a') : a' ∈ A_{s'} and a ≁ a'});
29         s.backtrack ← UpdateBack(s, a);
30         if s.backtrack = s.done then
31             return true;
32     return false;
```

that has been followed since the last time the algorithm backtracks to s (i.e., t is not already in $s.\mathbf{done}$ – see line 18), the algorithm updates $s.\mathbf{current}[t]$ to include all threads t' that later execute a transition that is dependent on τ (see lines 26-28). Subsequently, the $s.\mathbf{backtrack}$ set

is updated exactly in the style of Algorithm 2 (see line 29). If $s.\mathbf{backtrack}$ becomes equal to $s.\mathbf{done}$ then **IsComplete** returns *true* and the exploration from s stops.

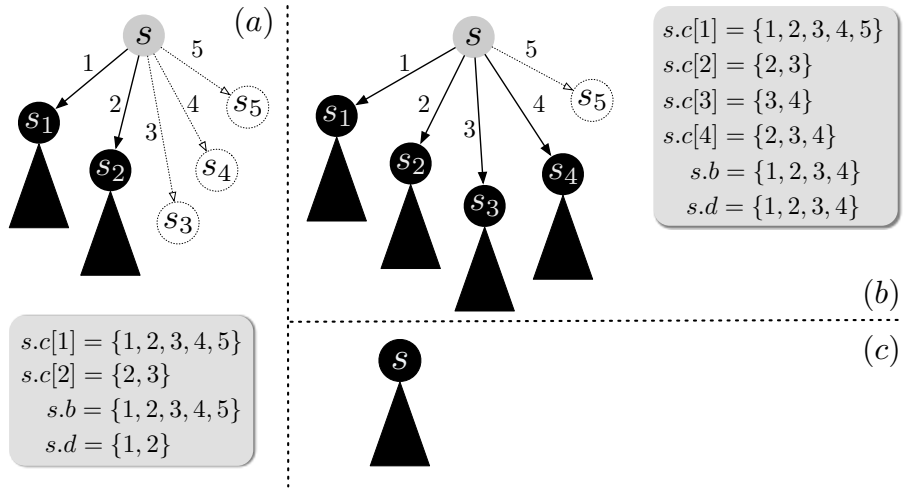

Figure 2.5: An example exploration of Lazy Source Set POR. We use the same conventions as in Figure 2.4.

We explain how Algorithm 3 works by an example. Figure 2.5(a) illustrates a scenario in which the exploration backtracks to a state s for the second time. After the first backtrack to s, the state space starting from the successor s_1 (resulted from following a transition of thread 1) was re-traversed in order to compute $s.\mathbf{current}[1]$. We assume that $s.\mathbf{current}[1]$ is changed to $\{1, 2, 3, 4, 5\}$ due to the dependent transitions encountered during this traversal. The set $s.\mathbf{backtrack}$ is set to $s.\mathbf{current}[1]$ as the latter contains all the enabled transitions. The exploration continues with a transition from s of thread 2, which is possible because thread 2 is in $s.\mathbf{backtrack} \setminus s.\mathbf{done}$. After backtracking to s for the second time, the re-traversal of the state space starting in s_2 leads to $s.\mathbf{current}[2] = \{2, 3\}$. The set $s.\mathbf{backtrack}$ remains the same after this computation. Then, in Figure 2.5(b), when backtracking to s for the fourth time, we assume that $s.\mathbf{current}[3] = \{3, 4\}$ and $s.\mathbf{current}[4] = \{2, 3, 4\}$. Since transitions of threads 2, 3, and 4 starting in s are independent of transitions of other threads that occur later, we can conclude that $\{2, 3, 4\}$ is a persistent set and $\{1, 2, 3, 4\}$ is a source set, and update $s.\mathbf{backtrack}$ to $s.\mathbf{done}$. Therefore, the exploration from s stops, as pictured in Figure 2.5(c). The set of transitions explored from s corresponds to a source set which is not a persistent set. The only persistent set that includes thread 1 is $\{1, 2, 3, 4, 5\}$.

2.2.2.1 Curbing Re-traversals for Efficiency

The re-traversals used to track dependencies in **current** sets are time-consuming, but several optimizations can be applied. The simplest optimization is not performing a traversal from the last

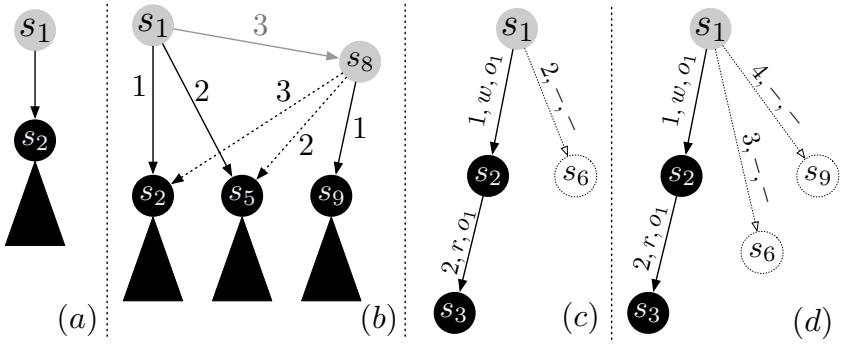

Figure 2.6: Optimizations to avoid re-traversing the state space. We use the same conventions as in Figure 2.4.

successor of s, called s', when backtracking to s. Then, $s.\mathbf{done}$ becomes equal to $safeSet(s)$ after adding the thread t leading to s' (see the first disjunct at line 21). This implies that no re-traversal is initiated for a state s that is reducible or it is irreducible but has only one enabled transition; for an example see Figure 2.6(a).

Second, a traversal from a successor of s does not have to be performed if after adding t to $s.\mathbf{done}$, all the threads enabled in s and not already in $s.\mathbf{done}$ lead to already visited states (see the second disjunct at line 21). The successors of those transitions may have not been pushed to the stack so they are added to the current LTS L_r (see line 22). This optimization also relies on initiating re-traversals of the state space only when backtracking from a new state (see the condition at line 11). Stopping the traversal is sound because there are no new states to explore from s and the exploration from s is already complete. A scenario where this optimization is enabled is shown in Figure 2.6(b). When backtracking to s_8 for the first time (from s_9), all the other threads enabled in s_8 lead to an already visited state. The re-traversal of the state space starting in s_9 is stopped since the exploration from s_8 is already complete (before returning, the transitions of threads 2 and 3 from s_8 are added to the current LTS).

While the optimizations above stop re-traversals altogether, several optimizations concern stopping a traversal early before reaching every state. In a concrete implementation, the declarative definitions at lines 27-28 translate to a (DFS) traversal of all the executions starting in s' and populating $s.\mathbf{current}[t]$ as new dependent transitions are found. This traversal can be stopped as soon as $s.\mathbf{current}[t]$ becomes "complete", i.e., it stores all threads in $safeSet(s)$. An example is shown in Figure 2.6(c) where the traversal starting in s_2 can stop immediately after the first transition since only threads 1 and 2 are enabled in s_1. Similarly, the traversal can be stopped immediately as $s.\mathbf{current}[t]$ contains a thread which is not enabled in s. In this case, $s.\mathbf{backtrack}$ will anyway be updated conservatively to include all the threads in $safeSet(s)$. An example is shown in Figure 2.6(d).

None of these optimizations are applicable in the eager version, or if they are, they are much harder to apply. Here, we take advantage of having to compute dependencies using forward traversals that explore full executions starting in a certain state as opposed to a backward traversal of states in the stack.

2.2.2.2 Soundness

The soundness of Algorithm 3, stated in the following theorem, is also based on proving that every state is expanded according to a source set. As in Theorem 1, it can be shown that $s.\mathbf{backtrack}$ is a source set for s when it becomes equal to $s.\mathbf{done}$. When backtracking to a state, the **current** sets satisfy the same specification as in the eager version.

Theorem 2. *Given a program represented by an LTS L, Algorithm 3 terminates with an LTS L_r that is sound for L.*

Proof. Similar to the proof of Theorem 1, it is enough to show that Proposition 2.1 holds. To end an exploration from a state s, **while** loop in line 4 of Algorithm 3 must be terminated. For this, **return** statement in line 15 must be reached and therefore, **IsComplete** in line 13 should return true. First, we show that **IsComplete** method eventually returns true. Due to to method $UpdateBack$, we know that $s.\mathbf{backtrack}$ can not contain a thread $t \notin safeSet(s)$. Hence, all the transitions of s that can be executed are only from threads in $safeSet(s)$ because of lines 5–8. Each time a transition of s is executed and then the search backtracks to s, $\mathbf{IsComplete}(s)$ is initiated. By the **for** loop in line 18, every transition from s that is executed is considered and by line 19, we know that all these transitions will be added to $s.\mathbf{done}$. Thus, $s.\mathbf{done}$ eventually becomes equal to $safeSet(s)$ which satisfies the condition in line 21 and as a result, **IsComplete** method returns true.

For **IsComplete** method to return true, either condition in line 21 or line 30 should be satisfied, where in both conditions $s.\mathbf{backtrack}$ must be equal to $s.\mathbf{done}$ before the return statement. That's why, equality of $s.\mathbf{done}$ and $s.\mathbf{backtrack}$ is the only condition for stopping an exploration from a state s. Similar to Algorithm 2, since $s.\mathbf{done}$ keep tracks of threads whose enabled transitions from s is already executed, the proof is deduced to showing that Proposition 2.2 holds for Algorithm 3 as well. The part between Proposition 2.2 and Proposition 2.3 in the proof of Theorem 1 applies totally the same and we show that T is a persistent set for s in L using the fact that L_r admits such an execution E' as it is concluded in the same proof. Assume by contradiction that T is not a persistent set for s. But as a result of backtracking to s after executing E' and invoking **IsComplete** method, t' will be added to $s.\mathbf{current}[t]$ (and eventually to T due to $UpdateBack$ method) in line 28 of Algorithm 3 since the transition label of τ' is an element of $A_{s'}$ ($t' \notin T$ and $act(\tau) \nsim act(\tau')$)

in line 27. Since it contradicts the assumption, T (and also $s.\mathbf{backtrack}$ by monotonicity of source sets) is a persistent set and a source set. □

2.2.3 Experimental Evaluation

We evaluate an implementation of the three algorithms S-POR, DE-S-POR, and DL-S-POR, presented in Section 2.2.1 and Section 2.2.2, in the context of the Java Pathfinder (JPF) model checker. As benchmark, we use bounded-size clients of Java concurrent data structures.

2.2.3.1 Implementation

We implement our algorithms as an extension of the `DFSHeuristic` class in JPF. To identify (in)visible actions (for computing safe sets), the only manual input is a list of class names that constitute the implementation of the concurrent data structure. The (in)visible transitions are automatically inferred from these class names and Java synchronization-related native methods used to implement compare-and-swap (CAS) for instance, which are all known. Every action reading or writing a field of an object in one of these classes, or which corresponds to a native method call are marked as visible (JPF makes it possible to parse the Bytecode instructions executed in a transition and determine the read/written object fields). Calls to the `lock` and `unlock` methods of a lock object are both considered as writes to the lock object, and therefore, visible. Any other action is considered as invisible. The dependency relation between visible actions is defined as usual, i.e., two actions that access the same object field, one of them being a write, are considered dependent. The way we define (in)visible actions is sound because the clients we consider do not contain additional computation. They simply call methods of the data structure (from different threads), the verification goal being related to combinations of return values observed in their executions.

2.2.3.2 Benchmarks

Our benchmark consists of bounded-size clients of 7 concurrent data structures from JDK8 or Synchrobench [46]: two set implementations based on *coarse-grain* and *fine-grain* locking, respectively (RWLockCoarseGrainedListIntSet and OptimisticListSortedSetWaitFreeContains), a set implementation based on a binary search tree and CAS, a wrapper on top of java.util.concurrent.ConcurrentLinkedQueue, java.util.concurrent.ConcurrentHashMap and a wrapper on top of it, and a hash map implementation based on coarse-grain locking. Since these implementations update shared memory using compare-and-swap or guarded by locks, they are data-race free and the restriction to sequential consistency is sound.

To evaluate our algorithms, we sampled 75 clients of these data structures where each client calls add and remove methods from 3 threads. Each thread contains up to 5 calls. We varied the contention on shared objects using less or more distinct inputs for add and remove methods.

We also use a number of buggy variations of the lock-based sets, RWLockCoarseGrainedListIntSet and OptimisticListSortedSetWaitFreeContains. We used Violat to generate client programs of these variations that admit consistency violations. Violat generates these client programs in three steps. First, Violat enumerates arbitrary test programs of a given data structure based on other inputs such as number of threads, maximum number of programs and so on. Next, it computes expected (ADT-admitted return-value) outcomes for each test program by computing and then recording the outcomes of all possible sequential executions. Finally, it runs the threads of each test program in parallel (using a stress testing tool or JPF), checks if the results are as expected, and reports the test programs that violates linearizability which is witnessed by observing an unexpected outcome.

To introduce bugs in the selected data structures before inputting them to Violat, we modify the placement of locks dynamically under certain conditions in certain methods (e.g., when the set contains a specific element). These conditions make it possible to control the difficulty of a bug. We consider four different classes of clients based on the number of invocations to methods that lead to bugs: (1) all of the invocations, (2) half of the invocations, (3) just a single invocation and (4) none of the invocations. We sampled 310 clients of these buggy variations with 3 threads and up to 4 calls per thread using Violat.

2.2.3.3 Results

We use S-POR, DL-S-POR, DE-S-POR to denote the three algorithms presented in this paper. For the same algorithms, we use JPF, DL-JPF, DE-JPF to represent the standard setup of JPF, and variations of the DL-S-POR and DE-S-POR when the safe set of a state s contains all the enabled threads in s ($safeSet(s) = enabled(s)$). The latter are used to evaluate the performance of the eager and lazy approaches while disabling the benefit of the static S-POR method. We compare implementations of S-POR, DL-S-POR and DE-S-POR between them, with JPF, DL-JPF and DE-JPF, with their stateless variations, and with a stateful variation of the optimal source set algorithm in [3] (called O-DPOR). For a fair comparison, we implement O-DPOR on top of S-POR without wakeup trees as their operations are quite expensive. The experiments were run on a 2,3 GHz Dual-Core Intel Core i5 processor with 8GB of RAM. We consider a timeout of 30 minutes.

Execution time comparison. Figure 2.7 and Figure 2.8 present a comparison in terms of execution time between different sets of algorithms. In Figure 2.7, we compare JPF, S-POR, DL-S-POR and DE-S-POR to observe the advantages of using our algorithms against the standard setup of

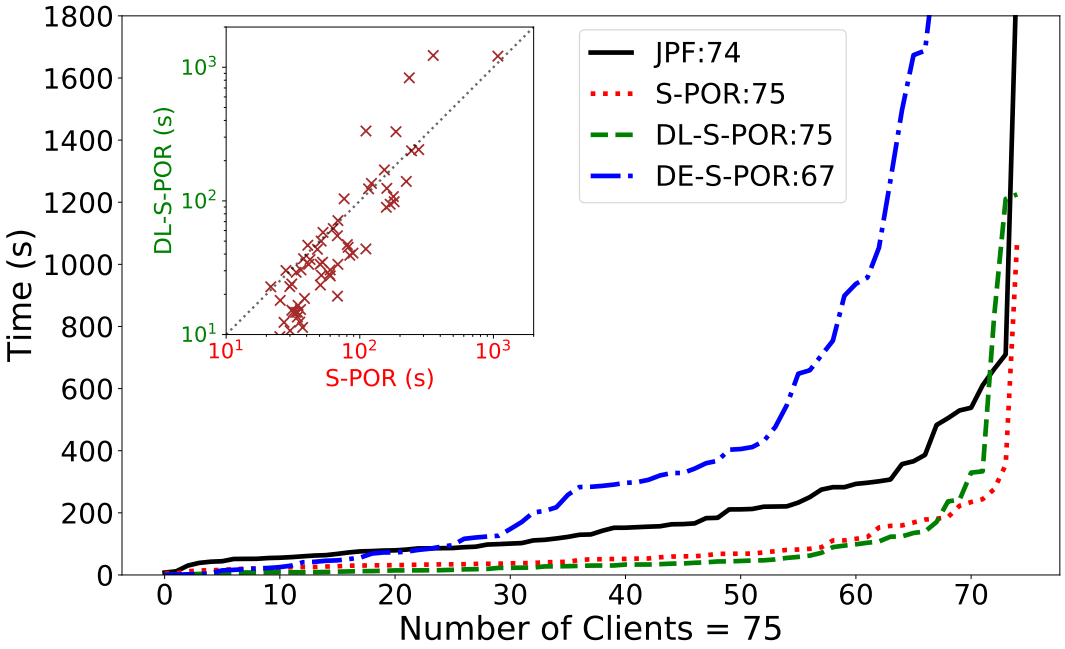

Figure 2.7: Quantile plot of running times for S-POR, DL-S-POR, DE-S-POR and JPF (for each algorithm, clients are ordered w.r.t. time in ascending order). The top left part shows a scatter plot for comparing S-POR and DL-S-POR.

JPF. In Figure 2.8, we compare DL-S-POR, DE-S-POR, DL-JPF and DE-JPF for investigating the gain by applying static filtering using S-POR as a baseline in dynamic algorithms. To ease the interpretation of the results, for each algorithm, we order clients according to execution time in ascending order. The numbers in the legend represent the number of clients on which a given algorithm terminates before the timeout. We omit O-DPOR because it times out for a large part of the benchmark, i.e., 39 out of the 46 clients on which it was run (our implementation of the algorithm in [3] does not support programs using locks which makes it inapplicable to the rest of the clients). This optimal algorithm manipulates happens-before constraints between steps in an execution, which results in a large overhead compared to our simpler tracking of pairwise dependencies. We also omit stateless variations of our algorithms since none of them finished before the timeout for any client. Note that stateless versions are obtained by disabling the state matching[1] in JPF, which also disables storing the full reachability graph.

Results based on Figure 2.7 show that the lazy source set computation in DL-S-POR gives a significant speedup w.r.t. DE-S-POR (and intuitively O-DPOR) while outperforming JPF. While S-POR processes few more clients faster w.r.t. DL-S-POR, the scatter plot on the top-left of Figure 2.7 shows that it is mostly in favor to DL-S-POR when clients are observed individually (this

[1]JPF uses hashing for state matching which is theoretically imperfect and can lead to incomplete results on rare occasions

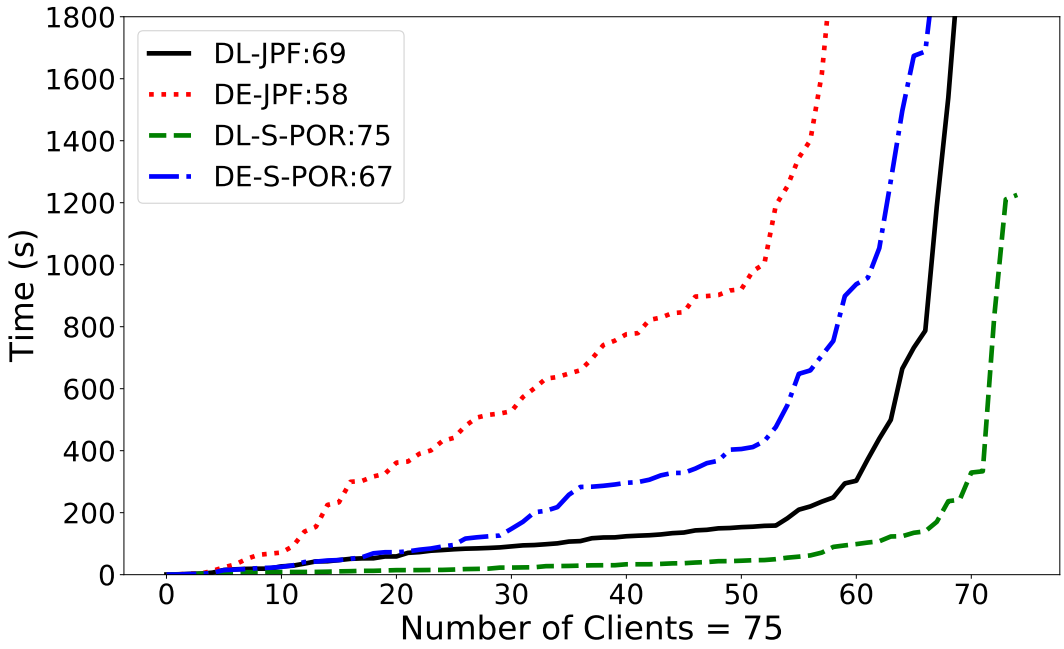

Figure 2.8: Quantile plot of running times for DL-JPF, DE-JPF, DL-S-POR and DE-S-POR (for each algorithm, clients are ordered w.r.t. time in ascending order as in Figure 2.7).

plot is given in logarithmic scale). DL-S-POR performs better than S-POR if there is a high potential for reduction, i.e., the ratio between the number of states explored by DL-S-POR over S-POR is smaller, and otherwise, S-POR is the best. This supports the hypothesis that if the potential for reduction is high enough then a carefully customized dynamic computation of source sets has a significant impact on performance. DL-S-POR gives an average speedup (average of speedups for each client) of 2.6 compared to S-POR. Overall picture suggests using a portfolio model checker where S-POR and DL-S-POR are run in parallel.

Similar to Figure 2.7, Figure 2.8 illustrates a comparison in terms of time between DL-S-POR, DE-S-POR, DL-JPF and DE-JPF. It shows that our algorithms outperforms their variations that are directly built on top of JPF (DL-S-POR against DL-JPF and DE-S-POR against DE-JPF). It also highlights the fact that the lazy approach is still better than the eager one even when the lazy approach is not based on S-POR.

Memory consumption comparison. Figure 2.9 presents a comparison in terms of memory consumption between S-POR and DL-S-POR, the most efficient algorithms according to Figure 2.7 and Figure 2.8, against the standard setup of JPF. We compared the maximum heap sizes using 74 clients that terminate before timeout for all algorithms. In all the experiments, the highest allocated heap size is 4.2GB. S-POR and DL-S-POR consume more memory than JPF because they have to store the transition labels which are used to reduce the explored state space. This

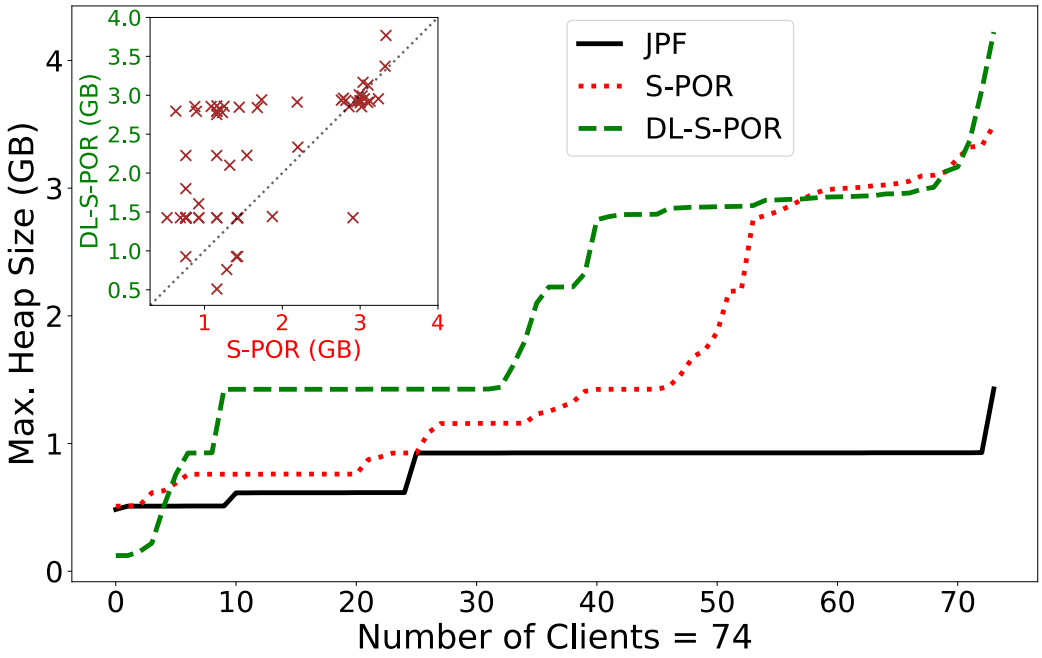

Figure 2.9: Quantile plot of memory consumption for S-POR, DL-S-POR, and JPF (for each algorithm, clients are ordered w.r.t. memory in ascending order). The top left part shows a scatter plot for comparing S-POR and DL-S-POR.

overhead is unavoidable for any form of dynamic partial order reduction. However, this memory consumption overhead is counterbalanced by significant speedups in terms of time. There is some memory overhead also due to storing the sets of transition labels manipulated by the algorithms, e.g., $s.\mathbf{current}$. But since these sets are maintained only for irreducible states and they are deleted for a state s when $s.\mathbf{done}$ equals $s.\mathbf{backtrack}$, their effects are not significant as storing transition labels.

For 32% of the clients, S-POR and DL-S-POR consume at most twice the memory consumed by JPF. For these clients, the average memory overhead is 1.00 for S-POR and 1.34 for DL-S-POR while the average speedup against JPF is 2.54 and 6.67, respectively. For 50% of the clients, S-POR and DL-S-POR consume in between 2 and 4 times the memory used by JPF. The average memory overhead for these clients is 2.20 for S-POR and 2.63 for DL-S-POR while the average speedup is 2.86 and 7.81, respectively. For the rest of the clients, the memory overhead is at most 7.79 and in average 4.11 for S-POR and 5.39 for DL-S-POR while the average speedup 3.28 and 5.31, respectively.

The top-left part of Figure 2.9 shows a pair-wise comparison of allocated maximum heap sizes in S-POR and DL-S-POR. These algorithms are incomparable in general. After investigating the clients individually, the results confirm that DL-S-POR consumes less memory than S-POR when

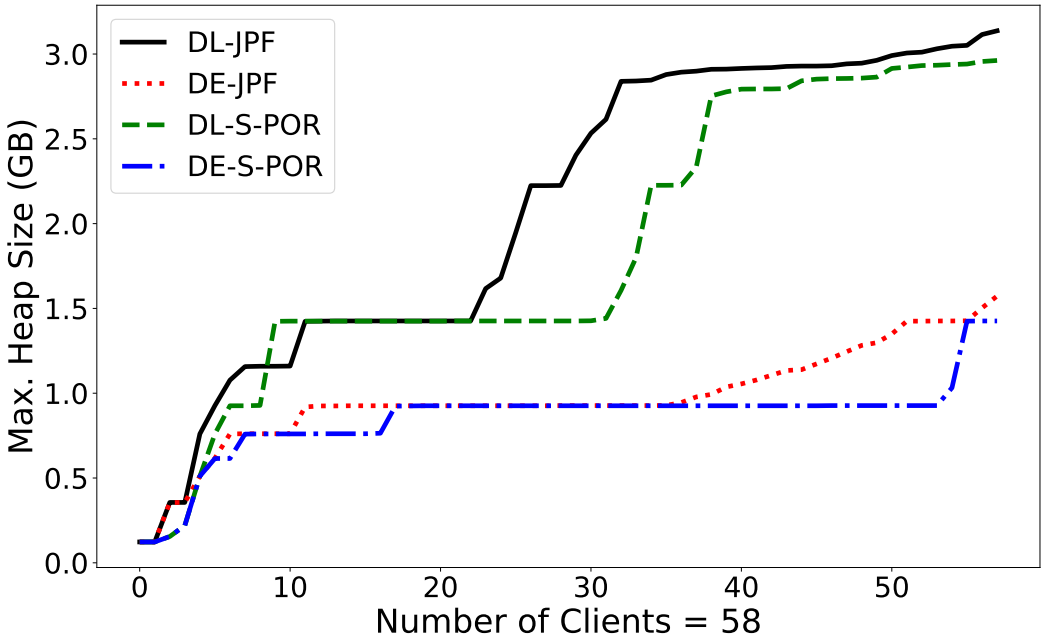

Figure 2.10: Quantile plot of memory consumption for DL-JPF, DE-JPF, DL-S-POR and DE-S-POR (for each algorithm, clients are ordered w.r.t. memory in ascending order as in Figure 2.9).

there is a high potential for reduction. The memory consumed for computing source sets is compensated by the reduction in the state space.

When we take a look at to Figure 2.10, it illustrates a comparison between DL-S-POR, DE-S-POR, DL-JPF and DE-JPF as in Figure 2.7, but in terms of memory consumption. As in Figure 2.9, we compared the maximum heap sizes of clients that finished before it timed out for all algorithms, which are 58 of them. It demonstrates that our algorithms are slightly better than their variations that are directly built on top of JPF. This memory overhead is mainly because of the additional transition labels that are not removed by the static filter. The overhead is also due to storing the sets of transition labels manipulated by the algorithms for all of the states rather than just for the irreducible ones but as mentioned previously, this overhead is negligible. This figure also shows that tracking dependencies with an eager approach does not increase the heap size as much as the lazy approach, although they explore the same state space. This difference in the memory overhead can not be explained by the memory that is used for storing the sets of transition labels or LTSs as they are all the same for both algorithms and they are kept in the same data structures. We suspect that this overhead can be due to low-level, internal details of JPF or due to the garbage collection process which might not keep up.

Transition enumeration. The performance of dynamic POR algorithms is generally affected by the order in which transitions starting in a certain state are enumerated. This order influences the

size of the computed persistent/source sets. This order is also important when enumerating states only until the first error is detected. We evaluate two strategies for defining this order, called *sequential* and *random*. For both of these strategies, the algorithm first selects a transition that leads to an already visited state, if one exists. We made this choice because it leads to better performance (this is adopted by the standard setup of JPF as well). In the sequential strategy, if there is no such transition, then the algorithm picks a transition by respecting a pre-defined order between the thread ids. In random, the next transition is selected uniformly at random.

We ran S-POR and DL-S-POR, the best algorithms as shown above, with all 6 permutations of the 3 threads for sequential and 3 different seeds for random. For each strategy, we report the average and minimum time over different instances as follows:

- The first set of experiments are for computing all reachable states, only with DL-S-POR. For S-POR, the enumeration strategy is not important since there is no dynamic computation of backtrack sets. We present separate figures for each different class of clients based on the number of invocations to methods that lead to bugs: (1) none of the invocations, (Figure 2.11), (2) just a single invocation (Figure 2.12) (3) half of the invocations (Figure 2.13) and (4) all of the invocations (Figure 2.14).
- The second set of experiments are for computing reachable states only until the first error, both with S-POR (Figures from 2.15 to 2.17) and DL-S-POR (Figures from 2.18 to 2.20). As the purpose is to find the first bug, clients in which none of the invocations are buggy, are not included in this set of experiments. Figures for the rest of the classes of clients are represented with the same order and visualization as in the first set. Differently from the first set of figures, the figures in second set are in logarithmic scale.

The results show that the random strategy performs better in average, shown on the left of Fig. 2.11–2.20, but worse w.r.t. minima, shown on the right of Fig. 2.11–2.20. The differences are more significant when enumerating states only until the first error as Fig. 2.15–2.20 reporting on this case are given in logarithmic scale. Thus, the sequential strategy should be preferred when using a portfolio model checker, i.e., parallel runs for each permutation of thread ids, and otherwise, the random strategy is better. This follows also from the average standard deviation being 28 seconds for random and 60 seconds for sequential, where the means are 17 and 20 seconds, resp. Note that, there is no significant impact observed from changing the algorithms or the number of buggy method invocations in the clients.

2.2.4 Related Work

Over the years various different techniques have been introduced to deal with the state explosion problem in model checking. For concurrent programs specifically, depth bounding [39], delay

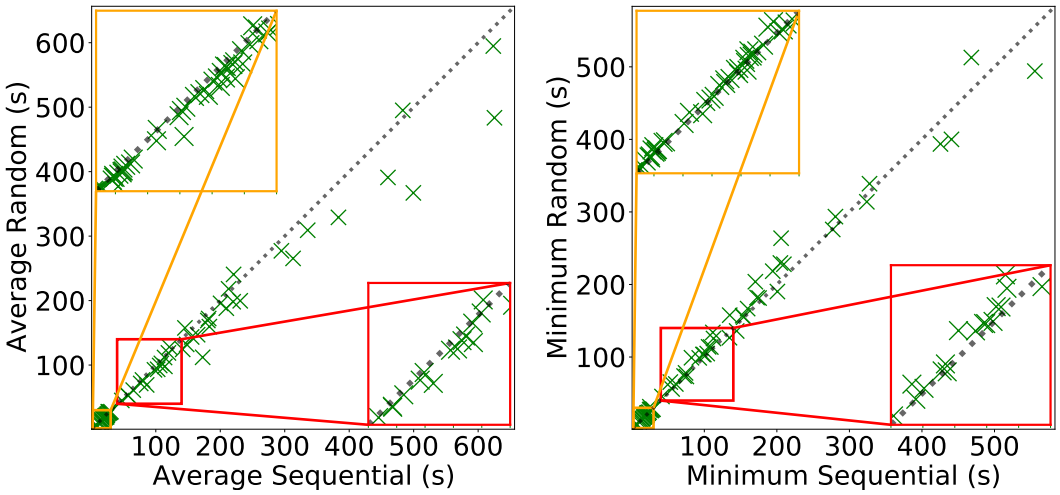

Figure 2.11: **ALL STATES \ DL-S-POR \ BUG: NO INVOCATION**
Time comparison between sequential and random strategies when DL-S-POR computes all states, using clients in which none of the invocations are buggy.

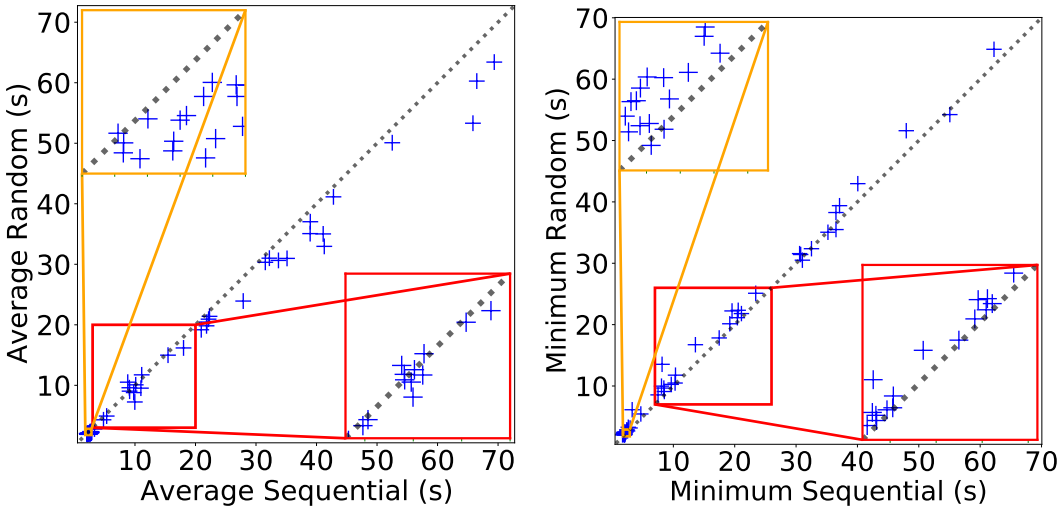

Figure 2.12: **ALL STATES \ DL-S-POR \ BUG: SINGLE INVOCATION**
Time comparison between sequential and random strategies when DL-S-POR computes all states, using clients in which only a single invocation is buggy.

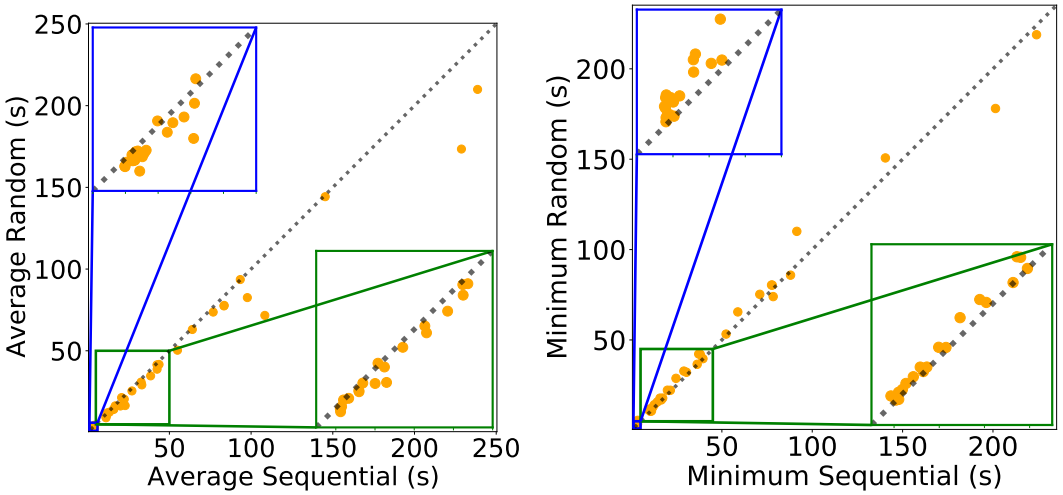

Figure 2.13: **ALL STATES \ DL-S-POR \ BUG: ½ OF THE INVOCATIONS**
Time comparison between sequential and random strategies when DL-S-POR computes all states, using clients in which half of the invocations are buggy.

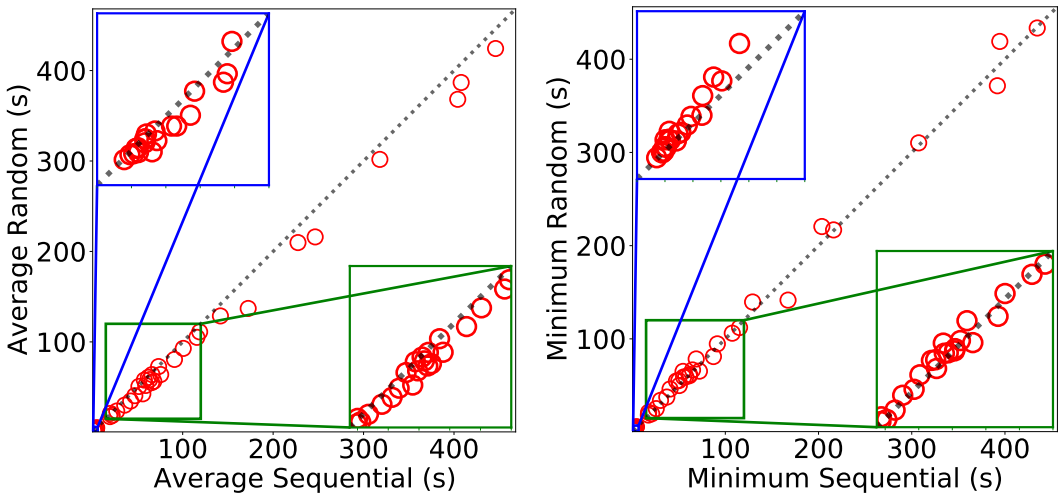

Figure 2.14: **ALL STATES \ DL-S-POR \ BUG: ALL OF THE INVOCATIONS**
Time comparison between sequential and random strategies when DL-S-POR computes all states, using clients in which all of the invocations are buggy.

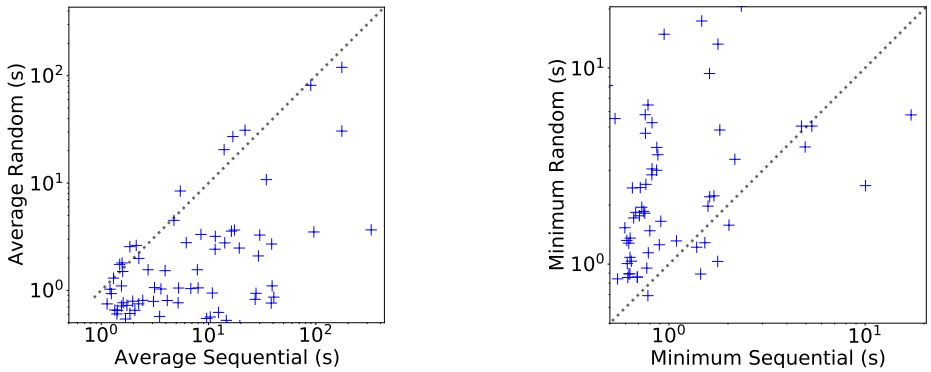

Figure 2.15: **FIRST ERROR \ S-POR \ BUG: SINGLE INVOCATION**
Time comparison (log scale) between sequential and random strategies when S-POR enumerates states only until the first error, using clients in which only a single invocation is buggy.

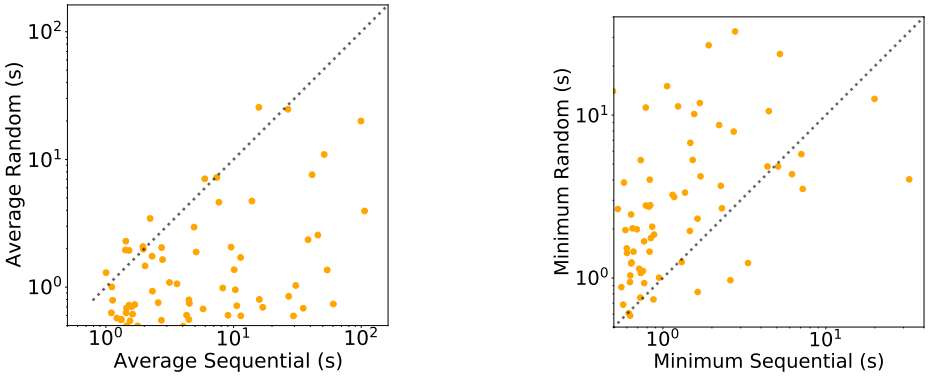

Figure 2.16: **FIRST ERROR \ S-POR \ BUG: ½ OF THE INVOCATIONS**
Time comparison (log scale) between sequential and random strategies when S-POR enumerates states only until the first error, using clients in which half of the invocations are buggy.

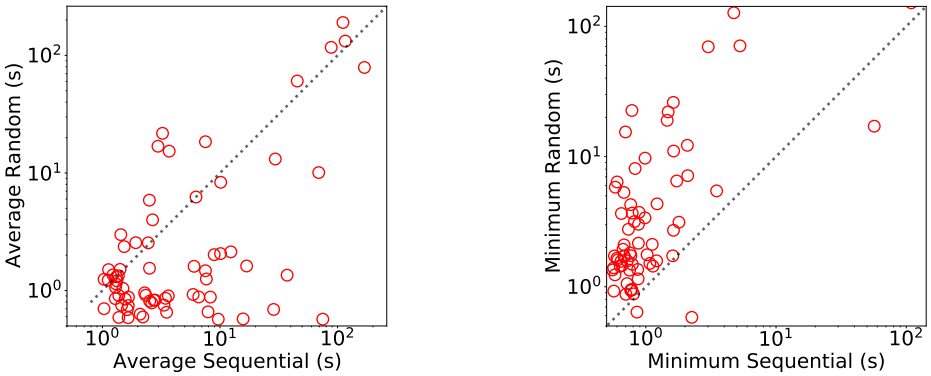

Figure 2.17: **FIRST ERROR \ S-POR \ BUG: ALL OF THE INVOCATIONS**
Time comparison (log scale) between sequential and random strategies when S-POR enumerates states only until the first error, using clients in which all of the invocations are buggy.

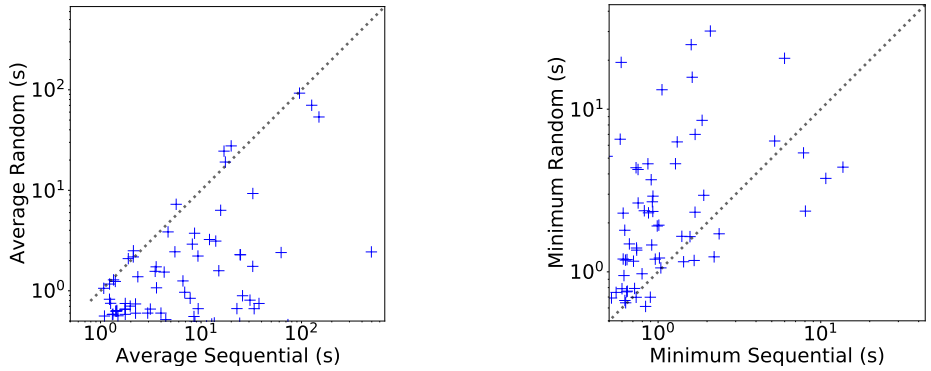

Figure 2.18: **FIRST ERROR \ DL-S-POR \ BUG: SINGLE INVOCATION**
Time comparison (log scale) between sequential and random strategies when DL-S-POR enumerates states only until the first error, using clients in which only a single invocation is buggy.

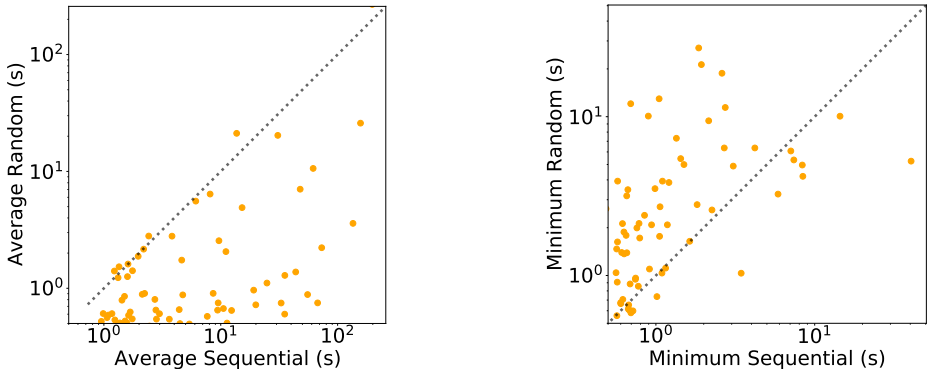

Figure 2.19: **FIRST ERROR \ DL-S-POR \ BUG: ½ OF THE INVOCATIONS**
Time comparison (log scale) between sequential and random strategies when DL-S-POR enumerates states only until the first error, using clients in which half of the invocations are buggy.

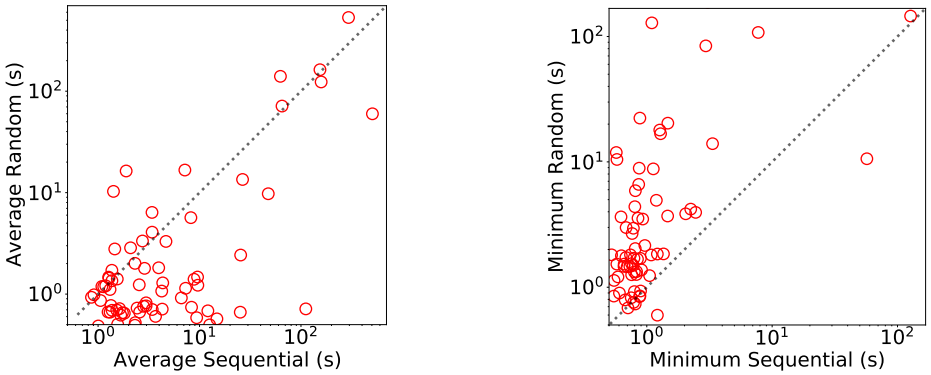

Figure 2.20: **FIRST ERROR \ DL-S-POR \ BUG: ALL OF THE INVOCATIONS**
Time comparison (log scale) between sequential and random strategies when DL-S-POR enumerates states only until the first error, using clients in which all of the invocations are buggy.

bounding [32], context bounding (bounding the number of context switches) [83], preemption bounding [75] and phase bounding [10] bring tractability to the model checking problem and have been shown to be effective for bug finding. These techniques are all *incomplete*, in the sense that lack of bugs does not guarantee the correctness of the system.

POR techniques reduce the search space by not exploring multiple executions from the same equivalence class, and are *complete*. Early techniques like ample sets [24, 51] and stubborn sets [41, 44] were based on static analysis. Sleep sets[41] were the first to guarantee optimality (one execution from each equivalence class) [42] by keeping track of information from the history of the exploration. However, they only prune transitions and cannot eliminate any state when used alone. Persistent sets [43, 60] generalized stubborn and ample sets and enabled development of dynamic POR (DPOR) methods.

In [35], an efficient stateful algorithm is proposed for computing persistent sets dynamically by considering currently explored parts of the state space. This algorithm needed large memory for keeping discovered states and the happens-before relation. The algorithm is improved in [103] with a more efficient state representation, and in [104] with a summary-based representation of the happens before.

In [68, 92], stateless dynamic POR techniques were introduced. Source sets [3] were introduced in the context of dynamic POR techniques such that the state space can be reduced up to the limit that is theoretically possible. They are generalizations of persistent sets and their relation with persistent sets are investigated in [2]. Our DE-S-POR and DL-S-POR algorithms rely on source sets but operate in the context of stateful model checking. The technique from [76] is similar to our S-POR algorithm for the GPU setting, but their choice of invisible actions is different than ours. While we focus on shared-memory programs running on top of a sequential consistency memory model, POR techniques have been also investigated in the context of weak memory models such as TSO or C11, e.g., [1, 4, 61, 62].

2.3 Root Causing Linearizability Violations

In this Section, we present an approach for identifying non-linearizability root-causes in a given execution, which equates root causes with optimal repairs that rule out the non-linearizable execution and as few linearizable executions as possible (from a set of linearizable executions given as input). Our approach can be extended to a set of executions and therefore in the limit, identify the root cause of the non-linearizability of a concurrent data structure as a whole. Sequential executions of a concurrent object are linearizable, and therefore, linearizability bugs can always be ruled out by introducing one atomic section per each method in the implementation of the corresponding object. Thus, focusing on atomic sections as repairs, there is a guarantee of existence of a repair in all scenarios. We emphasize the fact that our goal is to interpret such repairs as root-causes. Implementing these repairs in the context of a concrete concurrent object using synchronization primitives (eg., locks) is orthogonal and beyond the scope of this section. Some solutions are proposed in [56, 57, 99].

As a first step, we investigate the problem of finding *all optimal repairs* in the form of sets of atomic sections that rule out a given (non-linearizable) execution. A repair is considered optimal when roughly, it allows a maximal number of interleavings. We identify a connection between this problem and conflict serializability, an atomicity condition originally introduced in the context of database transactions. A repair that rules out a non-linearizable execution E can be obtained using a decomposition of the set of actions in E into a set of blocks that we call intervals, such that E is *not* conflict serializable with respect to this decomposition. Each interval will correspond to an atomic section in the repair (obtained by mapping events in the execution to statements in the code). A naive approach to compute all optimal repairs would enumerate all decompositions into intervals and check conflict-serializabiliy with respect to each one of them. Such an approach would be inefficient because the number of possible decompositions is exponential in both the number of events in the execution and the number of threads. We show that this problem is actually *polynomial time* assuming a fixed number of threads. This is quite non-trivial and requires a careful examination of the cyclic dependencies in non conflict-serializable executions. Assuming a fixed number of threads is not an obstacle in practice since recent work shows that most linearizability bugs can be caught with client programs with two threads only [28, 30].

In general, there may exist multiple optimal repairs that rule out a non-linearizable execution. To identify which repairs are more likely to correspond to root-causes, we rely on a given set of linearizable executions. We rank the repairs depending on how many linearizable executions they disable, prioritizing those that exclude fewer linearizable executions. This is inspired by the hypothesis that cyclic memory accesses ocurring in linearizable executions are harmless.

We evaluated this approach on several concurrent objects, which are variations of lock-based concurrent sets/maps from the Synchrobench repository. We considered a set of non-linearizable implementations obtained by modifying the placement of the lock/unlock primitives, and applied Violat to obtain client programs that admit non-linearizable executions. We applied our algorithms on the executions obtained by running these clients using JPF. Our results show that our approach is highly effective in identifying the precise root cause of linearizability violations since in every case, our tool precisely identifies the root cause of a violation that is discoverable by the client of the library used to produce the erroneous executions.

2.3.1 Overview

Shared variables:

```
range: integer initialized to 0
items: array of objects
     initialized to NULL
```

```
1  procedure push(x)
2    i := F&I(range);
3    items[i] := x
```

```
4  procedure pop()
5    t := range-1;
6    x := NULL;
7    for i := t downto 1 {
8      x := items[i];
9      items[i] := null;
10     if ( x != null ) break; }
11   return x;
```

Figure 2.21: A non-linearizable concurrent stack.

Figure 2.21 lists a variation of a concurrent stack introduced by Afek et al. [5]. The values pushed into the stack are stored into an unbounded array `items`; a shared variable `range` keeps the index of the first unused position in `items`. The `push` method stores the input in the array and it increments `range` using a call to an atomic fetch and increment (`F&I`) primitive. This primitive returns the current value of `range` while also incrementing it at the same time. The `pop` method reads `range` and then traverses the array backwards starting from the predecessor of this position, until it finds a position storing a non-null value. It also nullifies all the array cells encountered during this traversal. If it reaches the bottom of the array without finding non-null values, it returns that the stack is empty.

This concurrent stack is *not* linearizable as witnessed by the execution in Figure 2.22. This is an execution of a client with three threads executing two `push` and two `pop` operations in total. The `push` in the first thread is interrupted by operations from the other two threads which makes both `pop` operations return the same value `b`. The execution is not linearizable because the value `b` was pushed only once and it cannot be returned by two different `pop` operations.

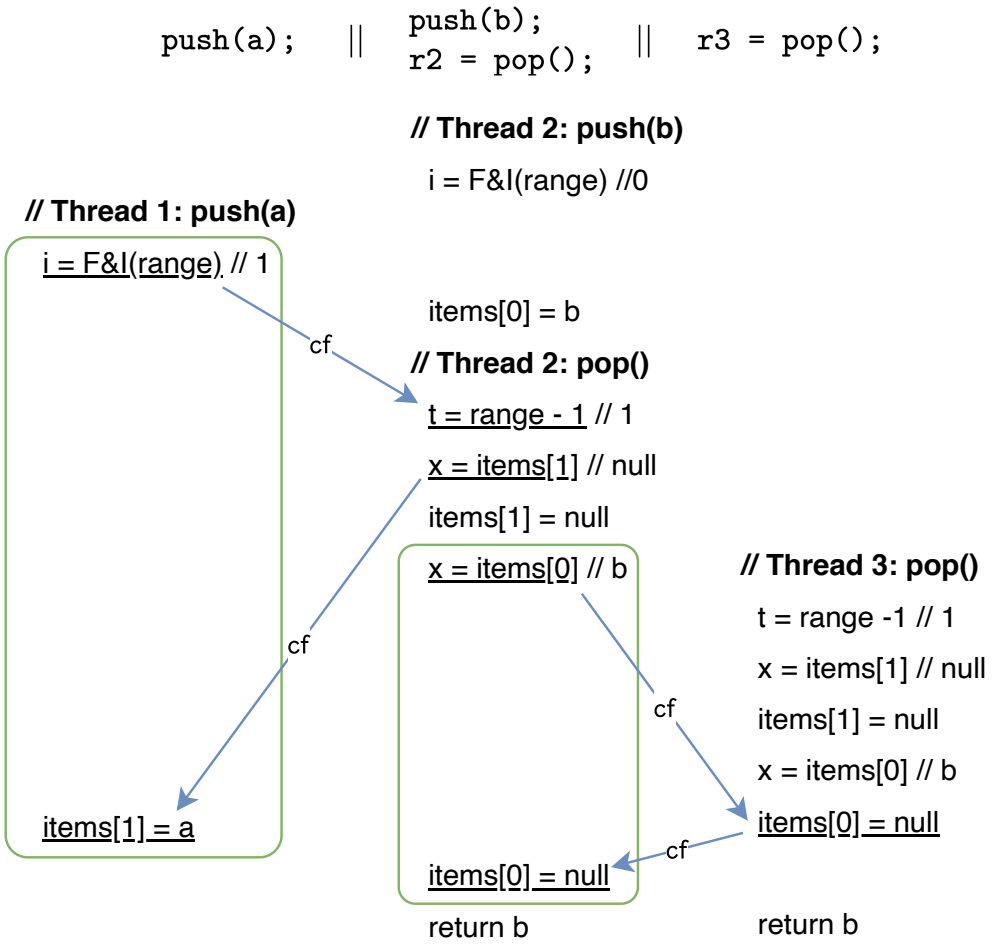

Figure 2.22: A client program of the concurrent stack of Figure 2.21 and one of its non-linearizable executions.

The root-cause of this violation is the non-atomicity of the statements at lines 8 and 9 of `pop`, reading `items[i]` and updating it to `null`. The stack is linearizable when the two statements are executed atomically (see [5]).

Our goal is to identify such root-causes. We start with a non-linearizable execution like the one in Figure 2.22. The first step is to compute all optimal repairs in the form of atomic sections that disable the non-linearizable execution. There are two such optimal repairs for the execution in Figure 2.22: (1) an atomic section containing the statements at lines 8 and 9 in `pop` (representing the root-cause), and (2) an atomic section that includes the two statements in the `push` method.

These repairs disable the execution because each pair of statements is interleaved with conflicting memory accesses in that execution. This is illustrated by the boxes that includes these two pair of statement and `cf` edges. `po` edges are already implied by the vertical alignment of actions. In

Section 2.3.3, we formalize this by leveraging the notion of conflict serializability. The execution is not conflict-serializable assuming any decomposition of the code in Figure 2.21 into a set of code blocks (transactions) such that one of them contains one of these two pairs. These repairs are optimal because they consist of a single atomic section of minimal size (with just two statements). We formalize a generic notion of optimality in Section 2.3.2 through the introduction of an order relation between repairs, defined as component-wise inclusion of atomic sections and compute the minimal repairs w.r.t. this order.

At the end of the first phase, our approach produces a set of all such optimal (incomparable) repairs. To isolate one as the best candidate, we use a heuristic to rank the optimal repairs. The heuristic relies on the hypothesis that repairs which disable fewer linearizable executions are more likely to represent the best candidate for the true root-cause of a linearizability bug.

For instance, the client in Figure 2.22 admits a linearizable execution where the first two threads are interleaved exactly as in Figure 2.22 and where the `pop` in the third thread executes *after* the first two threads finished. This is linearizable because the `pop` in the third thread returns the value `a` written by the `push` in the first thread in `items[1]` (this is the first non-null array cell starting from the end). Focusing on the two optimal repairs mentioned above, enforcing only the atomic section in the `push` will disable this linearizable execution. The atomic section in the `pop`, which permits this execution, is ranked higher to indicate it as the more likely root-cause. This is the expected result for our example.

This ranking scheme can easily be extended to a set of linearizable executions. Given a set of linearizable executions, we rank optimal repairs by keeping track of how many of the linearizable executions each disables.

2.3.2 Linearizability Violations and Their Root Causes

Given a non-linearizable library, our goal is to identify the root cause of non-linearizability in the library code. Let us start by formally describing the state space of all such causes and state some properties of the space that will aid the understanding of our algorithm. First, our focus is on a specific category of causes, namely those that can be removed through the introduction of new atomic code blocks to the library code without any other code changes.

Definition 3 (Non-linearizability Root Cause)**.** *For a non-linearizable library $\mathcal{L}$, the root cause is formally identified by $\mathcal{R}$, a set of atomic blocks $\mathcal{A}$ such that $\mathcal{L}$ is linearizable with the addition of blocks from $\mathcal{A}$.*

Observe that the set of atomic blocks identified in Definition 3 can conceptually be viewed as blocks of code whose non-atomicity is the *root cause* of non-linearizability and their introduction

would *repair* the library. For the rest of this section, we use the two terminologies interchangeably since for this specific class, the two notions perfectly coincide. The immediate question that comes to mind is whether Definition 3 is general enough. Observe that since linearizability is fundamentally an atomicity type property for individual methods in a library, if every single method of the library is declared atomic at the code level, then the library is trivially linearizable. The only valid executions of the library are the linear (sequential) executions in this case. Therefore,

Remark 1. *Every non-linearizable library can be made linearizable by adding atomic code blocks in $\mathcal{R}$ according to Definition 3.*

Since there always is a trivial repair, one is interested in finding a *good* one. The quality of a repair is contingent on the amount of *parallelism* that the addition of the corresponding atomic blocks removes from the executions of an arbitrary client of the library. Generally, it is understood that the fewer the number of introduced atomic blocks and the shorter their length, the more permissive they will be in terms of the parallel executions of a client of this library. This motivates a simple formal subsumption relationship between repairs of a bug. We say an atomic code block b subsumes another atomic code block b', denoted as $b \sqsupseteq_c b'$, if and only if b' is contained within b.

Definition 4 (Repair Subsumption)**.** *A repair $\mathcal{R}$ subsumes another repair $\mathcal{R}'$, we write $\mathcal{R} \sqsupseteq_c \mathcal{R}'$ if and only if for all atomic blocks $b' \in \mathcal{R}'$, there exists an atomic block $b \in \mathcal{R}$ such that $b \sqsupseteq_c b'$.*

It is easy to see that $\sqsupseteq_c$ is a partial order, and combined with the finite set of all possible program repairs gives rise to the concept of a set of optimal repairs, namely those that do not subsume any other repair. It can be lifted to sets of repairs in the natural way: $\mathbb{R} \sqsupseteq_c \mathbb{R}'$ if and only if $\forall \mathcal{R}' \in \mathbb{R}', \exists \mathcal{R} \in \mathbb{R} : \mathcal{R} \sqsupseteq_c \mathcal{R}'$.

Remark 2. *The set of executions of a library $\mathcal{L}$ with a repair $\mathcal{R}$ is a superset of the set of executions of $\mathcal{L}$ with the repair $\mathcal{R}'$ if $\mathcal{R}' \sqsupseteq_c \mathcal{R}$.*

This means that an optimal repair identification according to Definition 4 should lead to an optimal amount of parallelism in the library repaired by forcing the corresponding code blocks to execute atomically. The goal of our algorithm is to identify such a set of *optimal repairs*.

Now, let us turn our attention to an algorithmic setup to solve this problem. The non-linearizability of a library $\mathcal{L}$ is witnessed by a non-empty set of non-linearizable executions $\mathbb{E}$. These are the concrete erroneous executions of (a client of) the library, for which we intend to identify the repair.

Note that if E is a non-linearizable execution, then all the executions $E' \in [E]$ (that are equivalent to E) are also non-linearizable. Indeed, if E' is equivalent to E, then the values that are read

in E' are the same as in E[1], which implies that the return values in E' are the same as in E, and therefore, E' is non-linearizable when E is.

Consider a conceptual oracle, $\mathcal{O}^{\mathcal{L}}(\mathbb{E})$, that takes a set of non-linearizable executions of a library $\mathcal{L}$ and produces the set of all optimal repairs $\mathbb{R}$ such that each $\mathcal{R} \in \mathbb{R}$ excludes all the executions that are equivalent to those in $\mathbb{E}$. Then the following iterative algorithm produces $\mathbb{R}$ for a library $\mathcal{L}$:

1. Let $\mathbb{E} = \emptyset$ and $\mathbb{R} = \emptyset$.
2. Check if $\mathcal{L}$ with the addition of atomic blocks from $\mathbb{R}$ is linearizable:
   - Yes? Return $\mathbb{R}$.
   - NO? Produce a set of non-linearizability witnesses $\mathbb{E}'$ and let $\mathbb{E} = \mathbb{E} \cup \mathbb{E}'$.
3. Call $\mathcal{O}^{\mathcal{L}}(\mathbb{E})$ and update[2] the set of repairs $\mathbb{R}$ with the result.
4. Go to back to step 2.

Proposition 1. *The above algorithm produces an optimal set of repairs $\mathbb{R}$ that make its input library linearizable.*

It is easy to see that if oracle $\mathcal{O}^{\mathcal{L}}(\mathbb{E})$ can be relied on to produce perfect results, then the algorithm satisfies a progress property in the sense that $\mathbb{R}_{k+1} \sqsupseteq_c \mathbb{R}_k$, where $\mathbb{R}_k$ is the value of $\mathbb{R}$ in the k-th iteration of the loop. Following Remark 1, this chain of increasingly stronger repairs is bounded by the specific repair in which every method of the library $\mathcal{L}$ has to be declared atomic. Therefore, the algorithm converges. The assumption of optimality for $\mathcal{O}^{\mathcal{L}}(\mathbb{E})$ implies that on the iteration that the algorithm terminates, it will produce the optimal $\mathbb{R}$.

Note that in oracle $\mathcal{O}^{\mathcal{L}}$, the focus shifts from identifying the source of error for the entire library to identifying the source of error in a specific set of non-linearizability witnesses. First, we propose a solution for implementing $\mathcal{O}^{\mathcal{L}}$ for a singleton set, i.e. precisely one erroneous execution, and later argue why the solution easily generalizes to finitely many erroneous executions.

2.3.2.1 Repair Oracle Approximation

Given an execution E as a violation of linearizability, we wish to implement $\mathcal{O}^{\mathcal{L}}$ that takes a single execution E and proposes an optimal set of repairs for it.

Observe that if every execution of $\mathcal{L}$ is conflict serializable (i.e., equivalent to a sequential execution), assuming method boundaries as transaction boundaries, then it is necessarily linearizable.

[1] As a remark, we assume that program instructions are deterministic, which is usually the case.
[2] See Section 2.3.2.2 for more detail

Therefore, knowing that it is not linearizable, we can conclude that there exists some execution of $\mathcal{L}$ which is not serializable. Following the same line of reasoning, we can conclude that the erroneous execution E itself is not conflict serializable, for some choice of transaction boundaries. This observation is the basis of our solution for approximating repairs for non-linearizability through an oracle that is actively seeking to repair for non-serializability violations.

Definition 5 (Execution Eliminator)**.** *For an erroneous execution (a bug) E, a set of atomic blocks $\mathcal{R}$ is called an* execution eliminator *if and only if every execution $E' \in [E]$ is not an execution of the new library with the addition of blocks from $\mathcal{R}$.*

Any execution eliminator that removes E as a valid execution of a client of the library $\mathcal{L}$ (and all the executions in $[E]$), by amending the library for the conflict serializability violation, (indirectly) eliminates it as a witness to non-linearizability as well. Note that the universes of *execution eliminators* and *non-linearizability repairs* are the same set of objects, and therefore the subsumption relation $\sqsupseteq_c$ is well defined for execution eliminators, and the concept of optimality is similarly defined. Moreover, Definition 5 is agnostic to linearizability and can be interchangeably used for serializability repairs.

Theorem 3. *$\mathcal{R}$ is an execution eliminator for E if and only if E is not conflict serializable with transaction boundaries that subsume $\mathcal{R}$ (statements that are not included in the atomic sections from $\mathcal{R}$ are assumed to form singleton transactions).*

Proof. For the if direction, assume by contradiction that $\mathcal{R}$ is not an execution eliminator for E. This implies that there exists an execution $E' \in [E]$ where the sequences of actions corresponding to the atomic sections in $\mathcal{R}$ occur uninterrupted (not interleaved with other actions). This is a direct contradiction to E not being conflict serializable when transaction boundaries are defined precisely by the atomic sections in $\mathcal{R}$ because, if E' is cf-equivalent to its serial execution, then E must be cf-equivalent to the same serial execution as well, which is not the case. For the only if direction, assume by contradiction that E is conflict serializable. By definition, there is an equivalent execution E' where the sequences of actions corresponding to the atomic sections in $\mathcal{R}$ occur uninterrupted. Therefore, the library $\mathcal{L}'$ obtained by adding the atomic code blocks in $\mathcal{R}$ admits E', which contradicts the fact that $\mathcal{R}$ is an execution eliminator for E. □

The relationship between the set of execution eliminators for E and $\mathcal{O}^{\mathcal{L}}(E)$ can be made precise. Since every execution eliminator is a linearizability repair by definition, but not necessarily an optimal one, we have:

Proposition 2. *Let $\mathcal{O}^{\mathcal{L}}(E)$ represent the optimal set of repairs that eliminate E as a witness to non-linearizability and $\mathbb{R}$ be the set of optimal execution eliminators for E. We have $\mathbb{R} \supseteq \mathcal{O}^{\mathcal{L}}(E)$.*

This is precisely why the set of execution eliminators safely overapproximates the set of linearizability repairs for a single execution. Note that Theorem 3 links any execution eliminator (a set of code blocks) to a collection of dynamic (runtime) transactions. It is fairly straightforward to see that given the latter as an input, the former can be inferred in a way that the dynamic transactions generated by the static code blocks are as close as possible to the input transaction boundaries, assuming no structural changes occur in the code. In Section 2.3.3, we discuss how an optimal set of dynamic transaction boundaries can be computed, which give rise to a set of optimal execution eliminators.

2.3.2.2 Generalization to Multiple Executions

If we have an implementation for an oracle $\mathcal{O}^{\mathcal{L}}(E)$ that takes a single execution and produces the set of optimal execution eliminators for it, then the following algorithm implements an oracle for $\mathcal{O}^{\mathcal{L}}(\{E_1, \ldots, E_n\})$ for any finite number of executions:

1. Let $\mathbb{R} = \emptyset$.
2. For each E_i ($1 \leq i \leq n$): let $\mathbb{R}_i = \mathcal{O}^{\mathcal{L}}(E_i)$.
3. Let $\mathbb{X} = \mathbb{R}_1 \times \cdots \times \mathbb{R}_n$.
4. For each $\mathcal{X} \in \mathbb{X}$: let $\mathbb{R} = \mathbb{R} \cup \mathit{flatten}(\mathcal{X})$.
5. For each $\mathcal{R} \in \mathbb{R}$: if $\exists \mathcal{R}' \in \mathbb{R}$ s.t. $\mathcal{R} \sqsupseteq_c \mathcal{R}'$ then $\mathbb{R} = \mathbb{R} - \{\mathcal{R}\}$.

where $\mathit{flatten}(\mathcal{X})$ basically takes the union of repairs suggested by individual components of $\mathcal{X}$ while merging any overlapping atomic blocks. Note that the ith component of $\mathcal{X}$ suggests an *optimal* execution eliminator for E_i. If we want a tight combination of all such execution eliminators, we need the minimal set of atomic blocks that *covers* all atomic blocks suggested by each eliminator. Formally:

$$\mathit{flatten}(\langle \mathcal{R}_1, \ldots, \mathcal{R}_n \rangle) = \text{ smallest } \mathcal{R} \text{ wrt } \sqsupseteq_c \text{ st } \forall 1 \leq i \leq n : \mathcal{R} \sqsupseteq_c \mathcal{R}_i$$

we can then conclude:

Theorem 4. *If $\mathcal{O}^{\mathcal{L}}(E)$ produces the optimal set of execution eliminators for execution E, then the above algorithm correctly implements $\mathcal{O}^{\mathcal{L}}(\{E_1, \ldots, E_n\})$, that is, it produces the optimal set of repairs for the set of error executions $\{E_1, \ldots, E_n\}$.*

Proof. Assume by contradiction that $\mathcal{O}^{\mathcal{L}}(E)$ produces the optimal set of execution eliminators for execution E and there exist an optimal repair $\mathcal{R}' \notin \mathbb{R}$ (that is missed) where $\nexists \mathcal{R} \in \mathbb{R}$:

$\mathcal{R}' \sqsupseteq_c \mathcal{R}$. Note that, $\mathcal{R}' \notin \mathbb{R}$ is true at any point of the algorithm since by the contradiction assumption, it can not be removed from $\mathbb{R}$ after it is added. Then, there exist a $\mathcal{X}' \notin \mathbb{X}$ but, $\mathcal{X}' \in \mathbb{X}' = \mathbb{R}'_1 \times \cdots \times \mathbb{R}'_n$ where $\mathit{flatten}(\mathcal{X}') = \mathcal{R}'$. This shows that $\exists \mathbb{R}'_k : 0 < k \leq n \wedge \mathcal{R}' \sqsupseteq_c \mathbb{R}'_k$ and $\nexists \mathcal{R} \in \mathbb{R} : \mathcal{R} = \mathit{flatten}(\mathcal{X}) \wedge \mathcal{X} \in \mathbb{X} = \mathbb{R}_1 \times \cdots \times \mathbb{R}_n \wedge \mathbb{R}'_k \sqsupseteq_c \mathbb{R}_k$ which contradicts the assumption as $\mathcal{O}^{\mathcal{L}}(E_k)$ must have produced the optimal repair $\mathbb{R}'_k$. □

2.3.3 Conflict-Serializability Repairs

In this section, we investigate the theoretical properties of conflict serializability repairs to provide a set up for an algorithm that implements the oracle $\mathcal{O}^L$ for a single input execution. The goal of this algorithm is to take an execution E as an input and return the optimal execution eliminator for E, under the assumption that E witnesses the violation of linearizability.

2.3.3.1 Repairs and Conflict Cycles

We start by introducing a few formal definitions and some theoretical connections that will give rise to an algorithm for identifying an optimal set of atomic blocks that can eliminate an execution E as a witness to violation of conflict serialiazability.

Definition 6 (Decompositions and Intervals)**.** *A* decomposition *of an execution E is an equivalence relation D over its set of actions such that:*

- *D relates only actions of the same method invocation, i.e. if $(a_1, a_2) \in D$, then $mi(a_1) = mi(a_2)$, and*
- *the equivalence classes of D are continuous sequences of actions of the same method invocation, i.e., if $(a_1, a_3) \in D$ and $\{(a_1, a_2), (a_2, a_3)\} \subseteq \mathsf{po}_E$, then $\{(a_1, a_2), (a_2, a_3)\} \subseteq D$*

The equivalence classes of a decomposition D, denoted by $I_{E,D}$ are called intervals*.*

Observe that the relation $\sqsupseteq_c$ is well defined partial order over the universal all possible intervals (of all possible decompositions) of an execution E.

Definition 7 (Interval Graphs)**.** *Given an execution E, and decomposition D, an* interval graph *is defined as $G_{E,D} = (\mathsf{V}, \mathsf{E})$ where the set of vertices V is the set of intervals of D and the set of edges E is defined as follows*

$$\mathsf{E} = \{(i, i') |\, i \neq i' \wedge \exists a \in i, a' \in i' :\ (a, a') \in \mathsf{po}_E \cup \mathsf{cf}_E\}$$

Since, by definition, each edge in the interval graph is induced by an edge from either relation po_E or cf_E, but not both. We lift these relations over the sets of intervals in the natural way, that is:

$$(i, i') \in \mathsf{cf}^i_E \iff \exists a \in i, a' \in i' : \; a \neq a' \wedge (a, a') \in \mathsf{cf}_E$$
$$(i, i') \in \mathsf{po}^i_E \iff \exists a \in i, a' \in i' : \; a \neq a' \wedge (a, a') \in \mathsf{po}_E$$

Given an interval graph edge $(i, i') \in \mathsf{cf}^i_E \cup \mathsf{po}^i_E$, let

$$tre(i, i') = \{(a, a') \mid a \in i \wedge a' \in i' \wedge (a, a') \in \mathsf{cf}_E \cup \mathsf{po}_E\}$$

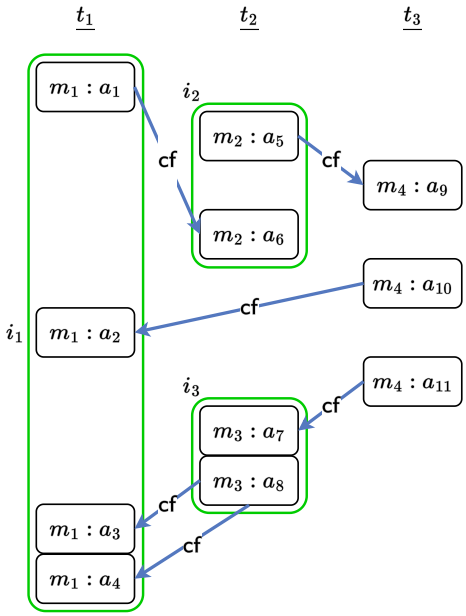

Figure 2.23: An interval graph.

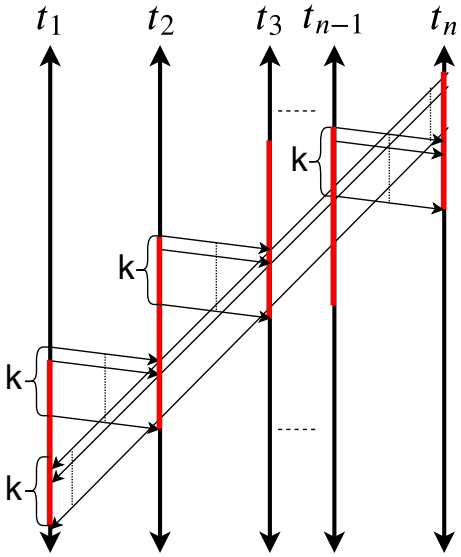

Figure 2.24: G^M_E with $|\mathsf{cf}_E|^{|\mathbb{T}|}$ chordless cycles.

Figure 2.23 illustrates an interval graph. Node $m_i : a_j$ denotes an action a_j of method invocation m_i, actions of the same thread are aligned vertically. As in Figure 2.22, we draw only cf_E edges since the po_E edges are implied by the vertical alignment of actions. Non-singleton intervals of D are $i_1 = \{a_1, a_2, a_3, a_4\}$, $i_2 = \{a_5, a_6\}$ and $i_3 = \{a_7, a_8\}$. Singleton intervals are identified by the corresponding action identifiers. Edges among interval nodes correspond to cf_E or po_E. For instance, $(i_1, i_2) \in \mathsf{cf}^i_E$ and $(i_2, i_3) \in \mathsf{po}^i_E$ since $(a_1, a_6) \in \mathsf{cf}_E$ where $a_1 \in i_1$, $a_6 \in i_2$, and $(a_6, a_7) \in \mathsf{po}_E$ where $a_7 \in i_3$, respectively. As an example for the function tre, we have $tre(i_2, i_3) = \{(a_5, a_7), (a_5, a_8), (a_6, a_7), (a_6, a_8)\}$ that consists of po_E edges and $tre(i_3, i_1) = \{(a_8, a_3), (a_8, a_4)\}$ that consists of cf_E edges.

For the degenerate decomposition in which each action is an interval of size one by itself, the interval graph collapses into an *execution graph*, denoted by G_E. Note that G_E is acyclic since the relations po_E and cf_E are consistent with the order between the actions in E.

Intervals are closely related to the static notion of transactions and the induced transaction boundaries on executions. For example, in the decomposition in which the intervals coincide with the boundaries of transactions (e.g. method boundaries), it is straightforward to see that the interval graph becomes precisely the *conflict graph* [34] widely known in the conflict serializability literature. It is a known fact that an execution is conflict serializable if and only if its conflict graph is acyclic [80]. Since E is not conflict serializable with respect to the boundaries of methods from $\mathcal{L}$, we know the interval graph with those boundaries is cyclic.

With intervals set as single actions, G_E is acyclic, and with the intervals set at method boundaries, it is cyclic. The high level observation is that there exist a decomposition D in the middle of this spectrum, so to speak, such that $G_{E,D}$ is cyclic, but $G_{E,D'}$ for any $D \sqsupset_c D'$ is acyclic. In the following we will formally argue why such a decomposition D is at the centre of identification of serializability repairs.

Additionally, a cycle in a graph is *simple* if only one vertex is repeated more than once. In graph theory, a *chord* of a simple cycle is an edge connecting two vertices in the cycle where these two vertices are not connected by a single edge in the cycle. A cycle is called *chordless* when it contains no chords (or chord edges).

Definition 8 (Critical Segment Sets)**.** *Let D be a decomposition such that the interval graph $G_{E,D}$ is cyclic and $\alpha = i_0 \dots i_{n-1}, i_0$ be a simple cycle. Define*

$$
\begin{aligned}
\mathit{edges}(\alpha) &= \mathit{tre}(i_0, i_1) \times \mathit{tre}(i_1, i_2) \times \cdots \times \mathit{tre}(i_{n-1}, i_0) \\
\mathit{segs}(\vec{e}) &= \{[a_k^{\odot}, a_k^{\otimes}] \mid 0 \leq k \leq n-1 \wedge \vec{e} \in \mathit{edges}(\alpha) \wedge (a_k^{\odot}, a_{(k+1)\ \mathit{mod}\ n}^{\otimes}) = \vec{e}_k\} \\
\mathit{critSegs}(\vec{e}) &= \{[a_k^{\odot}, a_k^{\otimes}] \in \mathit{segs}(\vec{e}) \mid (a_k^{\odot}, a_k^{\otimes}) \in \mathsf{po}_E\} \\
\mathit{CritSegs}(\alpha) &= \{s \mid \exists \vec{e} \in \mathit{edges}(\alpha) :\ s = \mathit{critSegs}(\vec{e})\}
\end{aligned}
$$

where the set $\mathit{CritSegs}(\alpha)$ is the set of all critical segments sets of cycle α.

Note that each cycle may induce several different segment sets, determined by $|\mathit{edges}(\alpha)|$. More importantly, each segment set includes at least one critical segment.

Lemma 1. *For any $\vec{e} \in \mathit{edges}(\alpha)$, we have $\mathit{critSegs}(\vec{e}) \neq \emptyset$.*

Proof. Assume by contradiction that $\vec{e} \in \mathit{edges}(\alpha)$ and $\mathit{critSegs}(\vec{e}) = \emptyset$ where $\alpha = i_0 \dots i_{n-1}$, i_0 is a simple cycle. Then, $\forall [a_k^{\odot}, a_k^{\otimes}] \in \mathit{segs}(\vec{e}) : 0 \leq k \leq n-1$, it must be $(a_k^{\odot}, a_k^{\otimes}) \notin \mathsf{po}_E$, which implies that every $a_k^{\otimes}$ occurs before $a_k^{\odot}$ ($a_k^{\otimes} < a_k^{\odot}$) as the intervals that contains these

actions (therefore, their thread ids) are already the same. Since $\forall(a_k^{\odot}, a_{(k+1) \bmod n}^{\otimes}) = \vec{e}_k \in \vec{e}$: $0 \leq k \leq n-1, \vec{e}_k \in \mathsf{po}_E \cup \mathsf{cf}_E$, we know that $a_k^{\odot} < a_{(k+1) \bmod n}^{\otimes}$. After combining these occurrence relations, we have $a_0^{\odot} < a_1^{\otimes} < a_1^{\odot} \ldots a_{n-1}^{\otimes} < a_{n-1}^{\odot} < a_0^{\otimes}$ which contradicts the assumption as $a_0^{\odot} < a_0^{\otimes}$ and therefore, $(a_0^{\odot}, a_0^{\otimes}) \in \mathsf{po}_E$. □

Example 1. *In Figure 2.23, $\alpha_1 = i_1, i_2, i_3, i_1$ is a simple cycle. Included in $edges(\alpha)$ are the following three example cycles and their corresponding segments:*

$$\alpha_1^1 = \langle (a_1, a_6), (a_6, a_7)(a_8, a_3) \rangle \qquad segs(\alpha_1^1) = \{[a_1, a_3], [a_6, a_6], [a_8, a_7]\}$$
$$\alpha_1^2 = \langle (a_1, a_6), (a_6, a_7), (a_8, a_4) \rangle \qquad segs(\alpha_1^2) = \{[a_1, a_4], [a_6, a_6], [a_8, a_7]\}$$
$$\alpha_1^3 = \langle (a_1, a_6), (a_5, a_8), (a_8, a_3) \rangle \qquad segs(\alpha_1^3) = \{[a_1, a_3], [a_5, a_6], [a_8, a_8]\}$$

The critical segments for these are $critSegs(\alpha_1^1) = \{[a_1, a_3]\}$, $critSegs(\alpha_1^2) = \{[a_1, a_4]\}$ and $critSegs(\alpha_1^3) = \{[a_1, a_3], [a_5, a_6]\}$.

There is a direct connection between the notion of critical segment sets and conflict serializability repairs that the following lemma captures. A segment is called *uninterrupted* in an execution E when all its actions occur continuously one after another in E without an interruption from actions of another interval.

Lemma 2. *Let α be a cycle in some interval graph $G_{E,D}$ of execution E which is not conflict serializable wrt. to the decomposition D and $critSeg_\alpha \in CritSegs(\alpha)$. There does not exist execution E' which is equivalent to E in which all segments from $critSeg_\alpha$ are uninterrupted in E'.*

Proof. Assume by contradiction that there exists an execution $E' \in [E]$ where the sequences of actions in all segments from $critSeg_\alpha$ are uninterrupted. As in Theorem 3, this is a direct contradiction to E not being conflict serializable because, if E' is cf-equivalent to its serial execution, then E must be cf-equivalent to the same serial execution as well. □

The immediate Corollary of Lemma 2 is that:

Corollary 1. *If one ensures the atomicity of the segments of actions in $CritSegs(\alpha)$ by adding atomic blocks at the code level, then E can no longer be an execution of the library. In other words, a set of such atomic code blocks is precisely an execution eliminator (Definition 5) for E.*

2.3.3.2 A Simple Algorithm

Lemma 2 and Corollary 1 suggest a simple enumerative algorithm to discover the set of all execution eliminators for a buggy execution E.

- Let $\mathbb{D}$ be the set of all decompositions of E and $\mathbb{R} = \emptyset$.

- For each $D \in \mathbb{D}$:
  - Let $\mathbb{C}$ be the set of all simple cycles in $G_{E,D}$.
  - For each $\alpha \in \mathbb{C}$:
    - Let $\mathbb{S} = \mathit{CritSegs}(\alpha)$.
    - $\mathbb{R} = \mathbb{R} \cup \mathbb{S}$
- For each $\mathcal{R} \in \mathbb{R}$:
  - If $\exists \mathcal{R}' \in \mathbb{R} : \mathcal{R} \sqsupseteq_c \mathcal{R}'$ then $\mathbb{R} = \mathbb{R} - \{\mathcal{R}\}$.

Theorem 5. *The above algorithm produces the optimal set of execution eliminators for a buggy execution E.*

This theorem is non-trivial, because the set of cycles considered are limited to simple cycles and an argument is required for why no optimal solution is missed as the result of this limitation. An important point is that any optimal execution eliminator $\mathcal{R}$ defines a decomposition D where the non-singleton intervals are precisely those defined by $\mathcal{R}$ such that $G_{E,D}$ contains a simple cycle α and the set of code blocks in $\mathcal{R}$ is a member of $\mathit{CritSegs}(\alpha)$. To prove, it is sufficent to show that the following lemma holds:

Lemma 3. *For every (non-simple) cycle α, there exist a simple cycle α' included in α such that $\mathit{CritSegs}(\alpha)$ subsumes $\mathit{CritSegs}(\alpha')$.*

Proof. This is true because for every cycle α, $\mathit{CritSegs}(\alpha)$ is equal to the union of $\mathit{CritSegs}(\alpha') : \alpha' \in \mathrm{A}'$ where A' is a set of simple cycles α' which are included in α. In other words, for every $\mathit{critSeg}_\alpha \in \mathit{CritSegs}(\alpha)$, there exist a simple cycle α' inside α such that $\mathit{critSeg}_\alpha \in \mathit{CritSegs}(\alpha')$. As simple cycles are minimal and there is no missing $\mathit{critSeg}$ when the considered set is just simple cycles, the above algorithm produces the optimal set of execution eliminators for a buggy execution E. □

Example 2. *The first loop of the above algorithm includes in $\mathbb{R}$ the execution eliminators induced by the critical segments mentioned in Example 1. After the last loop, however, only $\mathit{critSegs}(\alpha_1^1) = \{[a_1, a_3]\}$ will remain in $\mathbb{R}$ since the other two are subsumed by it.*

The algorithm is obviously very inefficient. There are two levels of enumeration: all decompositions and all cycles of each decomposition. Assuming that there are $O(|\mathsf{po}_E|)$ actions in a method invocation, then there are $O(2^{|\mathsf{po}_E|})$ different decompositions for it. Assuming that there are $O(|\mathbb{T}|)$ method invocations, we conclude that $|\mathbb{D}| = O(2^{|\mathsf{po}_E||\mathbb{T}|})$. There could be $O(2^{|E_E|})$ possible cycles for each decomposition where $E_E = \mathsf{po}_E \cup \mathsf{cf}_E$. Therefore, the first loop may

generate $O(2^{2|E_E||\mathbb{T}|})$ many repairs. The last loop iterates over $\mathbb{R}$ and each repair takes $O(\mathbb{R})$ time. The algorithm operates in time $O(2^{4|E_E||\mathbb{T}|})$. It is exponential both in the size of threads set and the graph. There are many redundancies in the output of the first loop, however. These are exploited to propose an optimized version of this algorithm.

2.3.3.3 A Sound Optimization

Consider an arbitrary cycle α in the interval graph $G_{E,D}$. If we want to trace the cycle α over the execution graph G_E, we would potentially need additional edges that would let us go against the program order inside some intervals that appear on α. Let us call the graph extended with such edges G_E^D. Formally, G_E^D includes all the nodes and edges from a execution graph and incorporates additional edges between the actions of each interval of D to turn it into a *clique*[1] which is by definition strongly connected and therefore accommodates the connectivity of any action of an interval to another action in it.

The converse also holds, that is, every *simple* cycle with at least one conflict edge in the G_E^D with the aforementioned additional edges corresponds to a cycle in the interval graph $G_{E,D}$. Note that the inclusion of at least one conflict edge is essential, since every interval graph cycle always includes one such edge by default; since the program order relation is acyclic. Formally:

Lemma 4. *For each simple cycle α of $G_{E,D}$, there exists a simple cycle α' of G_E^D that contains at most two actions from each interval in α.*

Proof. This lemma is trivially true since for any $\vec{e} \in \alpha$, $a_k^{\odot}$ and $a_k^{\otimes}$ of every $[a_k^{\odot}, a_k^{\otimes}]$ in $segs(\vec{e})$ are strongly connected in G_E^D which additionally contains every edge in $G_{E,D}$, including every $\vec{e}_k \in \vec{e}$. Thus, $\alpha' = a_0^{\odot}, a_0^{\otimes}, a_1^{\odot} \ldots a_{n-1}^{\odot}, a_{n-1}^{\otimes}, a_0^{\odot}$ is a valid simple cycle of G_E^D with at most two actions from each interval. □

The above lemma can immediately be generalized. Consider the graph G_E^M where M indicates the decomposition whose intervals coinciding with the library method boundaries. Since for any arbitrary decomposition D, we have $M \sqsupseteq_c D$, we can conclude that G_E^M includes all possible additional edges that one may want to consider as part of a cycle in an arbitrary G_E^D for an arbitrary decomposition D. Hence, the set of edges of G_E^M is a superset of the set of edges of all graphs G_E^D for all D. This immediately implies that the set of cycles of G_E^M is the superset of the set of cycles of all such graphs. This fact, combined with Lemma 4 leads us to the new simplified algorithm below in place of the one in Section 2.3.3.2:

- Let $\mathbb{R} = \emptyset$.

[1] A clique is a complete subgraph of a given graph.

- Let $\mathbb{C}'$ be the set of all simple cycles in G_E^M.
- For each $\alpha \in \mathbb{C}$:
  - Let $\mathbb{S} = critSegs(\alpha)$.
  - $\mathbb{R} = \mathbb{R} \cup \mathbb{S}$
- For each $\mathcal{R} \in \mathbb{R}$:
  - If $\exists \mathcal{R}' \in \mathbb{R} : \mathcal{R} \sqsupseteq_c \mathcal{R}'$ then $\mathbb{R} = \mathbb{R} - \{\mathcal{R}\}$.

Note that we are slightly bending the definition of $critSegs$ in the above algorithm, compared to the one given in Definition 8 since the input cycle there is formally a tuple, and here it is simply a list. The function is semantically the same, however and therefore we do not redefine it.

Observe that ever cycle of G_E^M corresponds to a cycle in some graph G_E^D for some decomposition D. This observation together with Lemma 4 and Theorem 5 implies the correctness of the above algorithm. Every cycle of every G_E^D is covered by the algorithm, and conversely every cycle considered is valid.

We can simplify the above algorithm one step further by further limiting the set of cycles $\mathbb{C}'$ that need to be enumerated.

Theorem 6. *The above algorithm produces the set of optimal execution eliminators for E if $\mathbb{C}'$ is limited to the set of simple chordless cycles of G_E^M.*

Theorem 6 makes a non-trivial and algorithmically subtle observation. Enumerating the set of all simple chordless cycles of G_E^D is a much simpler algorithmic problem to solve compared to the initial one from Section 2.3.3.2. Lemma 4 supports part of this argument since it ensures that all repairs explored in the algorithm from Section 2.3.3.2 are also explored by the above algorithm. For Theorem 6 to hold, it is enough to show that following Lemmas hold:

Lemma 5. *Critical segments of each cycles in G_E^M correspond to a valid (do not produce any junk) execution eliminator for E.*

Proof. This is trivially true because critical segments can include every action from the same thread as long as the method invocations of these actions are the same and this is the limit for the above algorithm as well. This is because of the fact that there is no additional edge $\notin \mathsf{po}_E \cup \mathsf{cf}_E$ introduced between actions of different methods. □

Lemma 6. *For every simple cycle α, there exist a simple chordless cycle α' included in α where $CritSegs(\alpha)$ subsumes $CritSegs(\alpha')$.*

Proof. This is correct because chord of a simple cycle (if it exists) can only be a cf edge and there always exists a chordless cycle α' included in α by following the chords recursively, which can only make the critical segments smaller. Thus, for every simple cycle α there always exists a chordless cycle α' such that $CritSegs(\alpha)$ subsumes $CritSegs(\alpha')$. □

When we combine Lemma 3 and 6, we have the following;

Corollary 2. *For every (non-simple) cycle α, there exist a simple chordless cycle α' included in α where $CritSegs(\alpha)$ subsumes $CritSegs(\alpha')$.*

In Section 2.3.4.1, we present an algorithm that solves the problem of enumerating $CritSegs(\alpha)$ for all simple chordless cycles $\alpha \in \mathbb{C}'$ effectively.

2.3.4 Repair List Generation

In this section, we first start by giving a detailed algorithm that produces the set of all optimal execution eliminators. These repairs suggest incomparable optimal ways of removing an erroneous execution from the library. We then present a novel heuristic that orders this set into a list such that the the ones ranked higher in the list are more likely to correspond to something that a human programmer would identify (amongst the entire set) as the ideal repair.

2.3.4.1 Optimal Repairs Enumeration Algorithm

In this section, we present an algorithm for enumerating all critical segments of simple chordless cycles $\alpha \in G_E^M$ with at least one cf_E edge. Next, we prove its correctness, and formally analyze its time complexity. The algorithm is the following:

- Let $\mathbb{C} = \emptyset$.
- For each sequence $\alpha = c_1, c_2, \ldots, c_n$ where $c_i \in \mathsf{cf}_E$ and $0 < n \leq |\mathbb{T}|$:
  - Let $c_i = (a_i^{\otimes}, a_i^{\odot})$ for all $i \in [1, n]$.
  - If $(a_i^{\odot}, a_{(i+1) \bmod n}^{\otimes}) \in \mathsf{E}_E^M \backslash \mathsf{cf}_E$ and $a_i^{\odot} \neq a_j^{\odot} : i, j \in [1, n]$ s.t. $i \neq j$:
    - $\mathbb{C} = \mathbb{C} \cup \{\alpha\}$

It enumerates all non-empty cf_E sequences of length less than or equal to $|\mathbb{T}|$. If the sequence forms a valid simple cycle and visits each thread at most once (i.e. there are no two distinct conflict edges such that its end points are on the same thread), then it is added to the result set $\mathbb{C}$. Correctness of the algorithm relies on combining Corollary 2 and the following observation:

Lemma 7. *If α is a chordless simple cycle of G_E^D with at least one cf_E edge, then α visits each thread at most once and it visits at least two threads.*

Proof. Since a cf_E edge is connecting two actions with different thread ids, if any cycle α contains at least one cf_E edge, it visits at least two threads with this edge. Now assume by contradiction that α visits a thread twice but it is a chordless simple cycle. Then, there will be at least two actions with the same thread, visited by two different cf_E edges. If these two actions are the same, this cycle is not simple. Else, there will be a path between these two actions by po_E edges and if this path is included in α, again it is not simple. Otherwise, this path is a chord and hence, it contradicts the assumption. □

As a corollary of Lemma 7:

Corollary 3. *A chordless cycle α could have at most $|\mathbb{T}|$ conflict edges.*

Proof. Otherwise, by the pigeon hole principle, at least two conflict edges end up in the same thread. □

Therefore, the algorithm can soundly enumerate only sequences of cf_E edges of length less than or equal to $|\mathbb{T}|$. Moreover, the choice of cf_E determines the rest of the edges in the cycle. Therefore, there are at most $O(|\mathsf{cf}_E|^{|\mathbb{T}|})$ chordless cycles with at least one cf_E edge of a graph G^D_E.

Note that, in general, the number of simple cycles can be exponential in the number of edges. This means that enumerating only chordless cycles reduces the size asymptotically. In other words, our proposed sound optimization of Section 2.3.3.3 is at the roof of the polynomial complexity results presented here.

Interestingly, this upper bound is not loose. There is a class of executions parametrized by $|\mathbb{T}|$ such that the number of chordless cycles with at least one cf_E edge is $|\mathsf{cf}_E|^{|\mathbb{T}|}$. Let $\mathbb{T} = \{t_1, \ldots, t_n\}$ be the set of threads and G^M_E has k parallel conflict edges between t_i and $t_{(i \bmod n)+1}$ for all $i \in [1, n]$. Moreover, conflict edges that start from t_i is above the conflict edges that end at t_i in terms of program order. This graph is depicted in Figure 2.24. To form a cycle, one needs to pick one of k edges between t_i and $t_{(i \bmod n)+1}$ for all $i \in [1, n]$. So, there are k^n cycles. Since $k = \frac{|\mathsf{cf}_E|}{|\mathbb{T}|}$, there are $\left(\frac{|\mathsf{cf}_E|}{|\mathbb{T}|}\right)^{|\mathbb{T}|}$ chordless cycles with a conflict edge. If we consider $|\mathbb{T}|$ as a constant, there are $\Omega(|\mathsf{cf}_E|^{|\mathbb{T}|})$ chordless cycles with at least one cf_E edge. We are finally ready to state the main complexity result:

Theorem 7. *Above enumeration algorithm generates all chordless cycles with at least one cf_E edge of G^D_E in $O((|\mathsf{po}_E| + |\mathsf{cf}_E|)|\mathsf{cf}_E|^{|\mathbb{T}|})$ time.*

Proof. The loop enumerates all the cf_E sequences of length at most $|\mathbb{T}|$ in $O(|\mathsf{cf}_E|^{|\mathbb{T}|})$ time. For each such sequence, it takes $O(|\mathsf{po}_E| + |\mathsf{cf}_E|)$ time to check whether this sequence forms a cycle

(if each consecutive conflict edges are connected through a $\mathsf{E}_E^M \setminus \mathsf{cf}_E$ edge) and whether it visits a thread more than once. As a consequence, the above bound holds. □

Lastly, there may be as many optimal repairs as there are chordless cycles in G_E^M. Consider the class of executions depicted in Figure 2.24. Each chordless cycle with at least one cf_E edge has exactly n critical segments (illustrated in red). Consider two distinct chordless cycles α_1 and α_2. There exists a thread t_i such that there is a different edge between t_i and $t_{(i \bmod n)+1}$ in α_1 compared to α_2. Without loss of generality, assume that the corresponding edge of α_1 has source and destination actions that appear before the source and destination actions of the corresponding edge of α_2 in program order (po_E). Then, α_1 has a larger critical segment on t_i and smaller critical segment in $t_{(i \bmod n)+1}$ compared to α_2. Therefore, the neither critical segment subsumes the other. Therefore, each chordless cycle with at least one cf_E edge produces an optimal repair.

This implies that the bound presented in Theorem 7, namely $O((|\mathsf{po}_E| + |\mathsf{cf}_E|)|\mathsf{cf}_E|^{|\mathbb{T}|})$, applies any other algorithm that outputs all optimal repairs.

2.3.4.2 Ranking Optimal Repairs

We argued through the example in Section 2.3.1 and a formal statement in Section 2.3.2.1 that not every eliminator of a buggy execution E is an optimal root cause for non-linearizability. All that we know is that they are all optimal execution eliminators. As a heuristic to identify optimal linearizability repairs out of a set of execution eliminators, we rely on another input in the form of a set Γ of linearizable executions, and rank execution eliminators depending on how many linearizable executions from Γ they disable, giving preference to execution eliminators that disable fewer ones. This heuristic relies on an experimental hypothesis that there are harmless cyclic dependencies that occur in linearizable executions.

Given a buggy execution E, and a set Γ of linearizable executions, we use the following algorithm to rank execution eliminators for E:

- Let $\mathbb{R}$ be the set of optimal execution eliminators for E
- For each $\mathcal{R} \in \mathbb{R}$:
  - Let $f(\mathcal{R}) = |\{E' \in \Gamma : \mathcal{R} \text{ is an execution eliminator for } E'\}|$
- Sort $\mathbb{R}$ in ascending order depending on $f(\mathcal{R})$ with $\mathcal{R} \in \mathbb{R}$.

Since the above algorithm is heuristic in nature, there are no theoretical guarantees for the optimality of its results. For instance, its effectiveness depends on the set of linearizable executions Γ given as input. We discuss the empirical aspects of the underlying hypothesis in more detail in Section 2.3.5.

2.3.5 Experimental Evaluation

We demonstrate the efficacy of our approach for computing linearizability root-causes on several variations of lock-based concurrent sets/maps from the Synchrobench repository. We consider three libraries from this repository: two linked-list set implementations, with coarse-grain and fine-grain locking, respectively, and a map implementation based on an AVL tree overlapping with two singly-linked lists, and fine-grain locking. We define three non-linearizable variations for each library by shrinking one atomic section only in the `add` method, only in the `remove` method, or an atomic section in each of these two methods. For each non-linearizable variation, we use Violat to randomly sample three library clients that admit non-linearizable executions[1]. We use Java Pathfinder [96] to extract all executions of each client, up to partial-order reduction, partitioning them into linearizable and non-linearizable executions. Executions are extracted as sequences of call/return actions and read/write accesses to explicit memory addresses, associated to line numbers in the source code of each of the API methods. The latter is important for being able to map critical segments (which refer to actions in an execution) to atomic code blocks in the source code.

In Table 2.1, we list some quantitative data about our benchmarks, the clients, and the non-linearizable variations identified by the line numbers of the modified atomic sections (the original libraries can be found in the Synchrobench repository). For instance, the first variation of RWLockCoarseGrainedListIntSet is obtained by shrinking the atomic section in the add method between lines [26, 32/35] to [32, 32/35] (there are two line numbers for the end of the atomic section because it ends with an `if` conditional).

For each non-linearizable execution E of a client C, we compute the set of optimal execution eliminators for E using the algorithm in Section 2.3.3.3 with the cycle enumeration described in Section 2.3.4.1. We then compute the ranking of these execution eliminators using as input the set of linearizable executions of C (the restriction to linearizable executions of the same client is only for convenience). Note that multiple execution eliminators can be ranked first since they disable exactly the same number of linearizable executions. Also, note that an optimal root-cause can disable a number of linearizable executions. This is true even for the ground truth repair (i.e. a repair that a human would identify trough manual inspection).

The results are presented in Table 2.2 and are self-explanatory. In the majority of cases, the first elements in this ranking are atomic sections which are precisely or very close to the expected results, i.e., atomic sections that belong to the original (error-free) version of the corresponding library. In some cases, the output of our approach is close, but not precisely the expected one. This is only due to the particular choice of the client used to generate the executions. In general, the quality of the produced repairs (compared to the ground truth) depends the types of behaviours

[1] These linearizability violations are quite rare. The frequencies reported by Violat in the context of a fixed client (when using standard testing) are in the order of 1/1000.

Table 2.1: Benchmark data. Column **Library** shows the transformation on the atomic section(s) of the original library (we write atomic sections as pairs of line numbers in square brackets), **Client** shows the clients (we abbreviate the names of `add` and `remove` to **a** and **r**, respectively.), **# n-lin.** (**# lin.** resp.,) gives the number of outcomes (set of return values) witnessing for non-linearizability (linearizability resp.,), similarly, **# bugs** and **#valid** give the number of non-linearizable and linearizable executions extracted using Java Pathfinder, respectively, **# ev.** and **# conf.** give the average number of actions and conflict edges in these executions, **Total(s)** and ***E* El.(s)** give the clock time in seconds for applying our approach, the latter excluding the Java Pathfinder time for extracting executions.

RWLockCoarseGrainedListIntSet (120 LOC)

Id	Library	Client	# lin.	# n-lin.	# bugs	# valid	# ev.	# conf.	Total(s)	*E* El.(s)
1	a: [26, 32/35] →[32, 32/35]	{r(0); r(1); a(1)} || {a(0); a(1); r(1)}	10	1	9	136	58	18	12	1
2		{a(0); a(1); r(1); a(1);} || {a(1); r(0); r(0); a(0)}	6	13	225	109	80	30	33	4
3		{a(1); a(1)} || {r(1); r(0)} || {a(0); r(1)}	4	12	60	325	57	36	266	4
4	r: [47, 54/56] →[53, 54/56]	{a(1); a(1); r(1)} || {r(0); a(0); r(1);}	3	1	18	37	56	18	10	1
5		{a(0); a(1); r(1); r(1)} || {a(1); r(0); r(1); a(1)}	22	3	54	319	80	34	24	4
6		{r(1); a(0)} || {a(1); r(1)} || {a(1); a(0)}	14	12	240	265	56	38	152	2
7	a: [26, 32/35] →[32, 32/35]	{a(1); r(1); a(1)} || {a(1); r(1); a(1)}	12	15	109	135	55	19	16	1
8		{r(1); a(1); r(0); r(0)} || {a(0); r(1); a(0); a(1)}	11	13	153	172	74	27	29	3
9	r: [47, 54/56] →[53, 54/56]	{a(1); r(1)} || { a(0); a(1)} || {r(0); r(1)}	13	7	168	301	57	38	485	7

OptimisticListSortedSetWaitFreeContains (193 LOC)

Id	Library	Client	# lin.	# n-lin.	# bugs	# valid	# ev.	# conf.	Total(s)	*E* El.(s)
1	a: [51, 52/56] →[52, 52/56]	{r(1); r(0); a(0)} || {a(1); r(1); a(0)}	6	3	27	91	96	37	10	2
2		{a(0); a(1); a(1); r(0)} || {a(0); a(1); r(1); a(1)}	11	22	243	232	153	75	54	11
3		{a(0); r(1)} || {a(0); a(1)} || {a(0); a(0)}	6	8	240	217	110	62	347	9
4	r: [78, 80/82] →[79, 80/82]	{r(0); a(0); r(0)} || {a(0); r(0); a(1)}	7	3	39	94	96	41	12	2
5		{r(1); a(0); r(1); r(1)} || {a(0); a(0); a(1); r(0)}	9	3	48	220	140	60	22	6
6		{a(1); r(0)} || {a(0); a(1)} || {r(0); r(1)}	13	3	92	541	109	95	457	10
7	a: [51, 52/56] →[52, 52/56]	{r(1); r(1); a(0)} || {a(1); r(0); r(1)}	6	4	39	97	92	33	11	3
8		{a(1); r(1); a(1); r(0)} || {r(0); r(1); a(1); r(0)}	6	4	24	55	115	44	12	1
9	r: [78, 80/82] →[79, 80/82]	{a(0); r(1)} || {a(0); a(1)} || {a(0); a(0)}	6	8	224	201	111	63	314	14

LogicalOrderingAVL (1088 LOC)

Id	Library	Client	# lin.	# n-lin.	# bugs	# valid	# ev.	# conf.	Total(s)	*E* El.(s)
1	a: [267, 291] →[268, 269] [274, 291]	{a(0,1); a(1,0);} || {r(0,1); a(1,0)}	4	1	40	70	127	43	75	3
2		{a(1,0); r(1,0); a(1,1)} || {a(0,0); a(1,0); r(1,0)}	7	1	27	279	209	77	233	16
3		{a(1,0); a(0,0); a(1,0); a(0,1)} || {a(0,0); a(0,1); r(0,1); r(1,1)}	10	1	15	213	207	85	179	8
4	r: [432, 451] →[433, 434] [436, 451]	{a(1,0); r(1,1); r(1,0)} || {a(0,1); r(1,0); r(0,0)}	2	1	54	112	168	64	120	5
5		{a(0,0); a(1,1); r(1,1)} || {r(1,1); a(1,0); r(1,1)}	5	2	84	229	183	83	146	12
6		{a(0,1); a(1,0); a(1,0); r(1,0)} || {a(1,1); a(0,0); a(1,0); r(1,0)}	22	6	132	769	234	115	379	40
7	a: [267, 291] →[268, 269] [274, 291] r: [432, 451] →[433, 434] [436, 451]	{a(1,1); a(1,0);} || {a(1,0); r(1,0)}	6	1	9	82	112	37	21	2
8		{a(0,0); a(1,1); r(1,1)} || {r(1,1); a(1,0); r(1,1)}	5	3	93	229	188	82	174	11
9		{a(1,1); a(0,1); a(1,1); r(0,1)} || {a(0,1); a(0,0); a(0,1); a(1,0)}	12	2	17	229	211	89	143	5

of the library that the client exercises. However, if our tool ranks repair $\mathcal{R}$ first, in the context of a client C, then after repairing the library according to $\mathcal{R}$ the client C produces no linearizability violations.

The methods in the libraries OptimisticListSortedSetWaitFreeContains and LogicalOrdering-AVL use optimistic concurrency, i.e., unbounded loops that restart when certain interferences are detected. This could potentially guide our heuristic in the wrong direction of giving the ground truth a lower rank. Indeed, a ground truth that concerns statements in the loop body could disable

Table 2.2: Experimental data. Column **#res** gives the number of different results (sequences of execution eliminators) returned by our algorithm when applied on each of the non-linearizable executions of a client, and ***E* El.** gives the first or the first two execution eliminators in the ranking obtained with our approach. For each execution eliminator we give the number of linearizable executions it disables (after →).

RWLockCoarseGrainedListIntSet

Id	Library	# res.	*E* El.
1	a: [26, 32/35] →[32, 32/35]	1	[[27, 35]] → 0
2		1	[[27, 35]] → 81
3		1	[[27, 35]] → 252
4	r: [47, 54/56] →[53, 54/56]	1	[[48, 54]] → 9
5		1	[[48, 54]] → 153
6		1	[[48, 54]] → 144
7	a: [26, 32/35] →[32, 32/35]	1	[[27, 35] [48, 54]] → 81
8		1	[[27, 35] [48, 54]] → 135
9	r: [47, 54/56] →[53, 54/56]	1	[[27, 35] [48, 54]] → 156

OptimisticListSortedSetWaitFreeContains

Id	Library	# res.	*E* El.
1	a: [51, 52/56] →[52, 52/56]	1	[[51, 56]] → 0
2		1	[[51, 56]] → 150
3		1	[[51, 56]] → 124
4	r: [78, 80/82] →[79, 80/82]	1	[[78, 80]] → 15
5		1	[[78, 80]] → 108
6		1	[[78, 80]] → 48
7	a: [51, 52/56] →[52, 52/56]	1	[[51, 56] [78, 80]] → 78
8		1	[[51, 56] [78, 80]] → 15
9	r: [78, 80/82] →[79, 80/82]	1	[[51, 56]] → 152

LogicalOrderingAVL

Id	Library	#res	*E* El.
1	a: [267, 291] →[268, 269] [274, 291] r: [432, 451] →[433, 434] [436, 451]	1	[[267, 290]] → 0
2		1	[[271, 279] [448, 451]] → 8
3		1	[[271, 279] [448, 451]] → 15
4		1	[[436, 451]] → 0
5		1	[[436, 451]] → 0
6		1	[[436, 451]] → 0
7		2	[[271, 279] [448, 451]] → 0 [[448, 454]] → 0
8		2	[[271, 279] [448, 451]] → 0 [[436, 451]] → 0
9		2	[[271, 279] [448, 451]] → 0 [[451, 454]] → 0

a large number of executions which only differ in the number of loop iterations. This, however, does not happen for small-size clients (like the ones used in our evaluation) since the number of invocations are bounded, which bounds the number of interferences and therefore the number of restarts.

Optimistic concurrency has the potential to mess with the heuristic, but this does not happen in small bounded clients as witnessed by our benchmark that does just fine.

To conclude, our empirical study demonstrates that given a good client (one that exercises the problems in the library properly), our approach is very effective in identifying the method at fault and the part of its code that is the root cause of the linearizability violation.

2.3.6 Related Work

Linearizability Violations. There is a large body of work on automatic detection of specific bugs such as data races, atomicity violations, e.g. [33, 89, 90, 97]. The focus of this paper is on linearizability errors. Wing and Gong [101] proposed an exponential-time monitoring algorithm for linearizability, which was later optimized by Lowe [71] and by Horn and Kroening [52]; neither

avoided exponential-time asymptotic complexity. Burckhardt et al. [12] and Burnim et al. [13] implement exponential-time monitoring algorithms in their tools for testing of concurrent objects in .NET and Java. Emmi and Enea [29, 30] introduce the tool Violat (used in our experiments) for checking linearizability of Java objects.

Concurrency Errors. There have been various techniques for fault localization, error explanation, counterexample minimization and bug summarization for sequential programs. We restrict our attention to relevant works for concurrent programs. More relevant to our work are those that try to extract simple explanations (i.e. root causes) from concurrent error executions. In [59], the authors focus on shortening counterexamples in message-passing programs to a set of "crucial actions" that are both necessary and sufficient to reach the bug. In [55], the authors introduce a heuristic to simplify concurrent error executions by reducing the number of context-switches. Tools that attempt to minimize the number of context switches, such as SimTrace [54] and Tinertia [55], are orthogonal to the approach presented in this paper. To gain efficiency and robustness, some works rely on simple patterns of bugs for detection and a simple family of matching fixes to remove them, e.g., [22, 56, 57, 81]. Our work is set apart from these works by addressing linearizability (in contrast to simple atomicity violation patterns) as the correctness property of choice, and by being more systematic in the sense that it enumerates all execution eliminators for a given linearizability violation. We also present crisp results for the theoretical guarantees behind our approach and an analysis of the time complexity. Weeratunge et al. [99] use a set of good executions to derive an atomicity "specification", i.e., pairs of accesses that are atomic, and then enforce it using locks.

There is large body of work on synchronization synthesis [8, 15, 16, 17, 23, 48, 73, 94, 95]. The approaches in [23, 95] are based on inferring synchronization by constructing and exploring the entire product graph or tableaux corresponding to a concurrent program. A different group of approaches infer synchronization incrementally from executions [94] or generalizations of bad executions [16, 17]. These techniques [16, 17, 94] also infer atomic sections but they do not focus on linearizability as the underlying correctness property but rather on assertion local violations. Several works investigate the problem of deriving an optimal lock placement given as input a program annotated with atomic sections, e.g., [21, 31, 106]. Afix [56] and ConcurrencySwapper [16] automatically fix concurrency-related errors. The latter uses error invariants to generalize a linear error execution to a partially ordered execution, which is then used to synthesize a fix.

Linearizability Repairs. Flint [70] is the only approach we know of that focuses on repairing non-linearizable libraries, but it has a very specific focus, namely fixing linearizability of composed map operations. It uses a different approach based on enumeration-based synthesis and it does not rely on concrete linearizability bugs.

3 Quorum Tree Abstractions of Consensus Protocols

In this chapter, we are interested in verifying consensus protocols such as Paxos, PBFT, Hot-Stuff, etc.. We propose a new abstraction for representing the executions of these consensus protocols that can be used in particular, to reason about their safety, i.e., ensuring *Agreement* and *Validity*. As mentioned before, (usually) protocol executions are composed of a number of communication-closed rounds, and each round consists of several phases in which a process broadcasts a request and expects to collect responses from a quorum of processes before advancing to the next phase. The abstraction is defined as a *sequential* object called *Quorum Tree (QTree)* which maintains a tree structure where each node corresponds to a different round in an execution. The operations of QTree, to add or change the status of a node, model quorums of responses that have been received in certain phases of a round.

For instance, a round in single-decree Paxos consists of two phases: a *prepare* phase where a predetermined leader broadcasts a request for joining that round and expects a quorum of responses from the other processes before advancing to a *vote* phase where it broadcasts a value to agree upon and expects a quorum of responses (votes) in order to declare that value as decided in that round. Rounds are initiated by their respective leaders and can run concurrently. The idea behind QTree is to represent a Paxos execution using a rooted tree where each node different from the root corresponds to a round where the leader has received *a quorum of responses in the prepare phase*. The parent-child relation models the data flow from one round to a later round: responses to join requests contain values voted for in previous rounds (if any) and one of them will be included by the leader in the vote phase request. The round in which that value was voted defines the parent. Then, each node has one out of three possible statuses: `ADDED` if the vote phase can still be successful (the leader can collect a quorum of votes) but this did not happen yet, `GHOST` if the vote phase can not be successful (e.g., a majority of processes advanced to the next round without voting), and `COMMITTED` if the leader has received a quorum of responses in the vote phase. This is a tree structure because before reaching a quorum in the vote phase of a round, other rounds can start and their respective leaders can send other vote requests (with possibly different values). The specific construction of requests and responses in Paxos ensures that all the `COMMITTED` nodes in

this tree belong to a *single* branch, which entails the Agreement property (this will become clearer when presenting the precise definition of QTree in Section 3.2).

The QTree abstraction is applicable to a wide range of protocols beyond the single-decree Paxos sketched above. It applies to state-machine replication protocols like Raft and HotStuf where the tree structure represents logs of commands (inputted by clients) stored at different processes and organized according to common prefixes (each node corresponds to a single command) and multi-decree consensus protocols like multi-Paxos [67] and its variants [47, 53, 65, 72], or PBFT where different consensus instances (for different indices in a sequence of commands) are modeled using different QTree instances.

We show that all these protocols are *refinements* of QTree in the sense that their executions can be intuitively, "linearized" to a sequence of operations on a QTree state, which are about agreeing on a branch of the tree called the *trunk*. These operations are defined as invocations of two methods *add* and *commit* for adding a new leaf to the tree (during which some other nodes may turn to `GHOST`) and changing the status of a node from `ADDED` to `COMMITTED`, respectively. Any sequence of invocations to these methods ensures that all the `COMMITTED` nodes lie on the same branch of the tree (the trunk). In relation to protocol executions, *add* and *commit* invocations that concern the same node correspond to receiving a quorum of responses in two specific phases of a round, which vary from one protocol to another.

The mapping between protocol executions and QTree executions is defined as in proofs of linearizability for concurrent objects with *fixed* linearization points. Analogous to linearizability, where the goal is to show that an object method takes effect instantaneously at a point in time called linearization point, we show that it is possible to mark certain steps of a given protocol as linearization points of *add* or *commit* operations[1], such that the sequence of *add* and *commit* invocations defined by the order between linearization points along a protocol execution is a correct QTree execution. We introduce a declarative characterization of correct QTree executions that simplifies the proof of the latter (see Section 3.3).

The QTree abstraction offers a novel view on the dynamics of classic consensus or state-machine replication protocols like Paxos, Raft, and PBFT, which relates to the description of recent Blockchain protocols like HotStuff and Bitcoin, i.e., agreeing on a branch in a tree. It provides a formal framework to reason uniformly about single-decree consensus protocols and state-machine replication protocols like Raft and HotStuff. For single-decree protocols (or compositions thereof), the parent-child relation between QTree nodes corresponds to the data-flow between a quorum of responses to a leader and the request he sends in the next phase while for Raft and HotStuff,

[1] These linearization points are *fixed* in the sense that they correspond to specific instructions in the code of the protocol, and they do not depend on the future of an execution. For an expert reader, this actually corresponds to a proof of strong linearizability [45].

it corresponds to an order set by a leader between different commands. It enables new correctness proofs that we believe are much more intuitive and "explainable" compared to classic proofs based on inductive invariants. An inductive invariant has to describe all intermediate states produced by all possible orders of receiving messages and a precise formalization is quite complex. As an indication, the Paxos invariant used in recent work [78] (see formulas (4) to (12) in Section 5.2) is a conjunction of eight quantified first-order formulas which are hard to reason about and not re-usable in the context of a different protocol. Compared to previous abstract specifications for reasoning about consensus protocols, e.g., [9, 36], QTree is designed to be less abstract so that the refinement proof, establishing the relationship between a given protocol and QTree, is less complex (see Section 3.9 for details). Also, we believe that QTree helps in improving the understanding of such protocols and writing correct implementations. For instance, our proofs are very clear about the phases of a round in which quorums need to intersect, which provides flexibility and optimization opportunities for deciding on quorum sizes in each phase. Depending on environment assumptions, quorum sizes can be optimized while preserving correctness.

3.1 Preliminaries

Similar to Section 2.1, we model a distributed (multiple processes in different machines) program implemented on top of message passing as a labeled transition system (LTS) $L = (\mathcal{S}, s_I, \Gamma, \mathcal{A})$, where

- A state in $\mathcal{S}$ is a tuple of process' local states and a set of messages in transit.
- The state $s_I \in \mathcal{S}$ is the unique initial state.
- $\mathcal{A}$ is a set of actions a where a is an indivisible action of a process such as receiving a set of messages, performing a local computation step, or sending a message.
- Γ is a set of labeled transitions (s, a, s') such that $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$.

As in Section 2.1, an execution E from a state s is a sequence of alternating states and actions. Trace τ is the sequence of actions projected from some execution E by respecting the program order and $\mathcal{T}(L)$ denotes all the traces of L.

For two LTSs L and L', we usually say that L is a refinement of L' when $\mathcal{T}(L) \subset \mathcal{T}(L')$. This refinement relation implies that any safety specification of L' holds for L as well as long as it can be stated on sequences of actions. Note that liveness properties can not be transferred since the inclusion concerns only finite traces.

We will consider a slight variation of this definition of refinement that applies to LTSs that do not share the same set of actions. This notion of refinement is parametrized by a mapping

Γ between actions of L and L', and it is defined by $\Gamma(\mathcal{T}(L)) \subset \mathcal{T}(L')$. Here, a mapping $\Gamma : A_L \to A_{L'}$ is extended to sequences and sets of sequences as expected, e.g., $\Gamma(a_1 \dots a_n) = \Gamma(a_1) \dots \Gamma(a_n)$. With this extension, the preservation of safety specifications requires certain constraints on the mapping Γ that will be discussed later.

3.2 Quorum Tree

We describe the QTree sequential object which operates on a tree and has two methods *add* and *commit* for adding a new node and modifying an attribute of a node (committing a node), respectively. When used as an abstraction of consensus protocols, invocations of these two methods correspond to certain quorums that are reached during a round of the protocol.

3.2.1 Overview

QTree is a sequential rooted-tree, a possible state being depicted in Figure 3.1. The nodes with black dashed margins are not members of the tree and they are discussed later. Each node in the tree contains a round number, a value, and a status field set to `ADDED`, `GHOST`, or `COMMITTED`. The round number acts as an identifier of a node since there can not exist two nodes with the same round number. The *Root* node is part of the initial state and its status is `COMMITTED`. A QTree state consists of a trunk, alive branches, and dead branches; a branch is a chain of nodes connected by the parent relation. Alive branches are extensible with new `ADDED` nodes but dead branches are not. The trunk is a particular branch of the tree that starts from the root. It contains all the `COMMITTED` nodes and it ends with a `COMMITTED` node. It may also contain `ADDED` or `GHOST` nodes. For example, in Figure 3.1, the trunk consists of *Root* and n_3. All alive branches are connected to the last `COMMITTED` node of the trunk (alive branches can include `ADDED` or `GHOST` nodes). For instance, in Figure 3.1, the subtree rooted at n_3 contains a single alive branch whose leaf node is n_5. Dead branches can contain only `GHOST` nodes. In Figure 3.1, the tree contains a single dead branch containing the node n_1.

Nodes can be added to the tree as leaves. The status of a newly added node is either `ADDED` or `GHOST`. The status `ADDED` may turn to `GHOST` or `COMMITTED`. The `GHOST` status is "final" meaning that it can never turn into `COMMITTED` afterwards. However, `GHOST` nodes can be part of alive branches, and they can help in growing the tree.

QTree has two methods *add* and *commit*:

- *add* generates a new leaf node with a round number r, value v, and parent p identified by the round number r_p given as input. Its status is set to `ADDED` or `GHOST`, provided that some conditions hold. If the status of the new node is set as `ADDED`, then it either extends

(has a path to the end of) an existing alive branch or creates a new alive branch from the trunk. The new node may also "invalidate" some other nodes by changing their status from `ADDED` to `GHOST`.

- *commit* extends the trunk by turning the status of a node from `ADDED` to `COMMITTED`. This extension of the trunk may prevent some branches to be extended in the future (some alive branches may become dead), i.e., future invocations of *add* that extend those branches will add only `GHOST` nodes.

Each node models the evolution of a round in a consensus protocol and the value attribute represents the value proposed by the leader of that round. The round and value attributes of a node are immutable and cannot be changed later. We assume that round numbers are strictly positive except for $Root$ whose round number is 0.

QTree applies uniformly to a range of consensus or state-machine replication protocols. We start by describing a variation that applies to single-decree consensus protocols, where a number of processes aim to agree on a single value. Multi-decree consensus protocols that are used to solve state-machine replication can be simulated using a number of instances of QTree, one for each decree (the instances are independent one from another). Then, state-machine replication protocols like HotStuff that rely directly on a tree structure to order commands can be simulated by the QTree for single-decree consensus modulo a small change that we discuss later.

3.2.2 Definition of the Single-Decree Version

Algorithm 4 lists a description of QTree in pseudo-code. The following set of predicates are used as conditions inside methods:

1. $link(n) \equiv n.\text{parent} \in \text{Nodes} \wedge n.\text{parent.round} < n.\text{round}$
2. $newRound(n) \equiv \forall n' \in \text{Nodes}.\ n'.\text{round} \neq n.\text{round}$
3. $maxCommitted(n) \equiv n.\text{status} = \texttt{COMMITTED} \wedge$
   $(\forall n' \in \text{Nodes}.\ n'.\text{status} = \texttt{COMMITTED} \implies n'.\text{round} < n.\text{round})$
4. $extendsTrunk(n) \equiv \exists n' \in \text{Nodes}.\ maxCommitted(n') \wedge$
   (n extends $n' \vee n.\text{round} < n'.\text{round}$)
5. $valid(n) \equiv link(n) \wedge newRound(n) \wedge extendsTrunk(n)$
6. $valueConstraint(n) \equiv n.\text{parent} \neq Root \implies n.\text{value} = n.\text{parent.value}$

Algorithm 4: The Quorum Tree object

```
1  Initialize:    /* ⊥ denotes non-initialized values */
2      Root.round = 0; Root.status = COMMITTED;
3      Root.value = ⊥; Root.parent = Root;
4      Nodes = {Root};
5  Method add (r, v, r_p)
6      Pre: r > 0
7      n = new Node(round = r, status = ⊥, value = v, parent = p : p.round = r_p);
8      if valid(n) ∧ valueConstraint(n) then
9          Nodes = Nodes ∪ {n};
10         n.status = ADDED;
11         if ∃n' ∈ Nodes. n'.round > n.round then
12             n.status = GHOST;
13         forall n' ∈ Nodes. n'.round < n.round do
14             if n is conflicting with n' then
15                 n'.status ← GHOST;
16         return OK
17     return FAIL
18 Method commit (r)
19     if ∃ n ∈ Nodes. n.round = r ∧ n.status = ADDED then
20         n.status ← COMMITTED;
21         return OK
22     return FAIL
```

The *add* method (lines 5-17) generates a new node n with round, value, and parent set according to the method's inputs. Then, it adds n to the tree by linking it to the selected parent if n satisfies the following *validity* conditions:

- n's parent belongs to the tree and its round number is smaller than r (predicate *link* at (1)),
- the tree does not contain another node with round number r (predicate *newRound* at (2)),
- if r is bigger than the round number of the last node of the trunk, then n must extend the trunk (predicate *extendsTrunk* at (4)),
- n's value must be the same as its parent's value unless the parent is the $Root$ (predicate *valueConstraint* at (6)).

The *valid* predicate at (5) is the conjunction of the first three constraints.

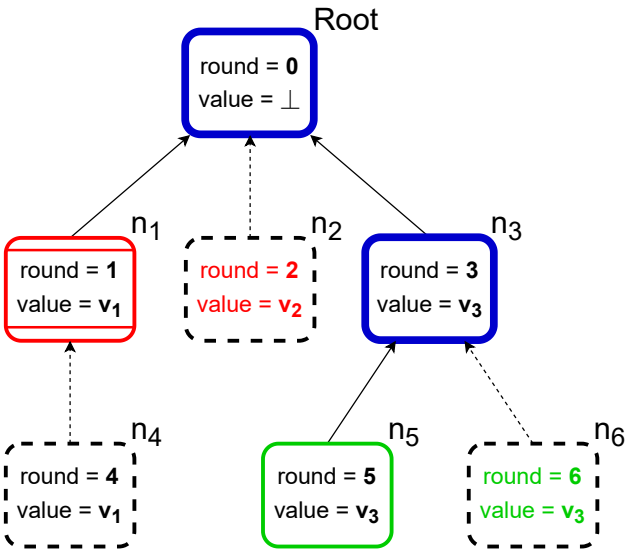

Figure 3.1: A state of QTree. We represent ADDED nodes with green solid margins, GHOST nodes with red double-line margins, and COMMITTED nodes with blue thick margins. The nodes with black dashed margins are not part of the state, they are fictitious nodes used to explain the method for adding new nodes.

For example, let us consider an invocation of add in a state of QTree that contains the non-dashed nodes in Figure 3.1. If the invocation generates n_2, n_4, or n_6 (receiving as input the corresponding attributes), then n_2 and n_6 do satisfy all these constraints and can be added to the tree. The node n_4 fails the *extendsTrunk* predicate because it is not extending the last node of the trunk (n_3) and its round number is higher.

If a node n satisfies the conditions above, the add method turns its status to either ADDED or GHOST. If there is another node in the tree with a higher round number, n's status becomes GHOST. Otherwise, it becomes ADDED. As a continuation of the example above, the status of n_2 is set to GHOST because the tree contains node n_3 with a higher round number and the status of n_6 is set to ADDED.

Moreover, the addition of n can "invalidate" some other nodes, turn their status to GHOST. This is based on a notion of *conflicting* nodes. We say that two nodes are conflicting if they are on different branches, i.e., there is no path from one node to the other. An add invocation that adds a node n changes the status of all the nodes n' in the tree that conflict with n and have a lower round number than n, to GHOST. For example, the top part of Figure 3.2 pictures a sequence of QTree states in an execution, to be read from left to right (for the moment, ignore the Paxos part). The first state represents the result of executing $add(1, v_1, 0)$ on the initial state of QTree, adding node n_1. Executing $add(3, v_2, 0)$ on this first state creates another node n_3 and sets its status to ADDED. This invocation will also turn the status of n_1 to GHOST since its round number

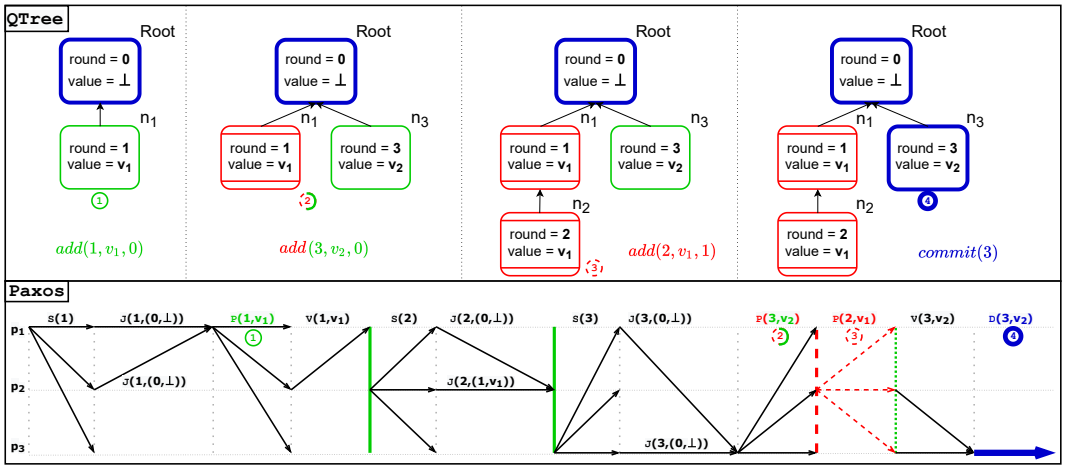

Figure 3.2: **Top:** QTree - Explaining the behavior of *add* and *commit* methods. Colors should be interpreted as in Figure 3.1.
Bottom: Paxos - A single-decree Paxos execution simulated by the QTree execution above. We abbreviate types of messages with their first letter, e.g., `START` with `S`, and payloads are given in parenthesis. Concerning the vertical lines, green solid ones represent the beginning of a new round, red dashed ones show a round that is lagging behind (delayed) and green dotted ones represent returning to a newest round from a delayed round.

is less than the round number of n_3 and they are on different branches. Afterwards, by executing $add(2, v_1, 1)$, a node n_2 is added to the tree with status `GHOST` since there is a node n_3 on a different branch which has a higher round number.

The method *add* returns OK when the created node is effectively added to the tree (it satisfies the conditions described above) and $FAIL$, otherwise.

Lastly, the *commit* method takes a round number r as input and turns the status of the node containing r to `COMMITTED` if it was `ADDED`. If successful, it returns OK and $FAIL$, otherwise. As a continuation of the example above, the top-right part of Figure 3.2 pictures a state obtained by executing $commit(3)$ on the state to the left. This sets the status of n_3 to `COMMITTED` as n_3 was previously `ADDED`. Note that the conditions in *add* ensure that the tree can not contain two nodes with the same round number.

Safety Properties. We show that the QTree object in Algorithm 4 can be used to reason about the safety of single-decree consensus protocols, in the sense that it satisfies a notion of *Validity* and *Agreement*. More precisely, we show that every state that is reachable by executing a sequence of invocations of *add* and *commit* (in Algorithm 4), called simply *reachable state*, satisfies the following:

- *Validity*: every node different from *Root* contains the same value as a child of *Root*, and
- *Agreement*: every two `COMMITTED` nodes different from *Root* contain the same value.

Proposition 3 (Validity)**.** *Every node in a reachable state that is different from Root contains the same value as a child of Root.*

Proof. A node n is added to the tree only if the predicate *valueConstraint* holds, which implies that it is either a child of $Root$ or it has the same value as its parent which is a descendant of $Root$. Also, since the value attribute of a node is immutable, any `COMMITTED` node contains the same value that it had when it was created by an add invocation. □

Therefore, the fact that a consensus protocol refining QTree satisfies validity, i.e., processes decide on a value proposed by a client of the protocol, reduces to proving that the phases of a round simulated by add invocations that add children of $Root$ use values proposed by a client. This is ensured using additional mechanisms, i.e., a client broadcasts its value to all participants in the protocol, so that each participant can check the validity of a value proposed by a leader.

Next, we focus on *Agreement*, and show that `COMMITTED` nodes belong to a single branch of the tree.

Proposition 4. *Let n_1 and n_2 be two `COMMITTED` nodes in a reachable state. Then, n_1 and n_2 are not conflicting.*

Proof. Assume towards contradiction that QTree reaches a state where two `COMMITTED` nodes n_1 and n_2 are conflicting. Let $r_1 = n_1$.round and $r_2 = n_2$.round. Without loss of generality, we assume that $r_1 < r_2$. Such a state is reachable if $add(r_1, \_, \_)$ and $add(r_2, \_, \_)$ resulted in adding the nodes n_1 and n_2 and set their status to `ADDED` (we use _ to denote arbitrary values), and subsequently, $commit(r_1)$ and $commit(r_2)$ switched the status of both n_1 and n_2 to `COMMITTED`. If $add(r_1, \_, \_)$ were to execute before $add(r_2, \_, \_)$, then $add(r_2, \_, \_)$ would have changed the status of n_1 to `GHOST` because it is conflicting with n_2. Otherwise, if $add(r_2, \_, \_)$ were to execute before $add(r_1, \_, \_)$, then the latter would have set the status of n_1 to `GHOST` since the tree contains n_2 that has a higher round number. In both cases, executing $commit(r_1)$ can never turn the status of n_1 to `COMMITTED`. □

Proposition 4 allows to conclude that any two `COMMITTED` nodes (different from $Root$) contain the same value. Indeed, a node can become `COMMITTED` only if it was `ADDED`, which implies that is has the same value as its parent (the predicate *valueConstraint* holds), and by transitivity, as any of its ancestors, except for $Root$.

Proposition 5 (Agreement)**.** *Let n_1 and n_2 be two `COMMITTED` nodes in a reachable state, which are different from Root. Then, n_1.value = n_2.value.*

3.2.3 State Machine Replication Versions

The single-decree version described above can be extended easily to a *multi-decree* context. As multi-decree consensus protocols, used in state machine replication, can be seen as a composition of multiple instances of single-decree consensus protocols, a multi-decree version of QTree is obtained by composing multiple instances of the single-decree version. Each of these instances manipulates a tree as described above without interference from other instances. The validity and agreement properties above apply separately to each instance.

The single-decree version can also be extended for state machine replication protocols like Hot-Stuff and Raft where the commands (values) are a-priori structured as a tree, i.e., each command given as input is associated to a predetermined parent in this tree. Then, the goal of such a protocol is to agree on a sequence in which to execute these commands, i.e., a branch in this tree. Simply removing the *valueConstraint* condition in the add method (underlined in Algorithm 4) enables QTree to simulate such protocols. A node's value need not be the same as its parent's value to be valid for add. Proposition 4 that implies the agreement property of such protocols still holds (Proposition 5 does not hold when the *valueConstraint* condition is removed; this property is specific to single-decree consensus). Since the value field remains immutable, the validity property of such protocols reduces to ensuring that the values generated during phases simulated by add correspond to commands issued by the client (Proposition 3 is also specific to single-decree consensus and it does not hold). As before, this requires additional mechanisms, i.e., a client broadcasting a command to all the participants in the protocol, whose correctness can be established quite easily.

3.3 Consensus Protocols Refining QTree

In the following, we show that a number of consensus protocols are refinements of QTree in the sense that their executions can be mimicked with add and $commit$ invocations. This is similar to a linearizable concurrent object being mimicked with invocations of a sequential specification. The refinement relation allows to conclude that the *Validity* and *Agreement* properties of QTree imply similar properties for any of its refinements.

The definition of the refinement relation relies on a formalization of protocols and QTree as labeled transition systems. For a given protocol, a state is a tuple of process local states and a set of messages in transit, and a transition corresponds to an indivisible step of a process (receiving a set of messages, performing a local computation step, or sending a message). For QTree, a state is a tree of nodes as described above and a step corresponds to an invocation to add or $commit$. An execution is a sequence of transitions from the initial state.

Refinement corresponds to a mapping between protocol executions and QTree executions. This mapping is defined as in proofs of linearizability for concurrent objects with *fixed lineariza-*

tion points, where the goal is to show that each concurrent object method appears to take effect instantaneously at a point in time that corresponds to executing a fixed statement in its code. Therefore, certain steps of a given protocol are considered as linearization points of add and $commit$ QTree invocations (returning OK), and one needs to prove that the sequence of invocations defined by the order of linearization points in a protocol execution is a correct execution of QTree. With respect to the terminology in Section 3.1, this is an effective definition for a mapping Γ between actions of the protocol and actions of QTree, i.e., $add/commit$ invocations such that Γ applied to protocol executions results in correct QTree executions. In the following, we provide a characterization of correct QTree executions that simplifies such refinement proofs.

An *invocation label* $add(r, v, r_p) \Rightarrow RET$ or $commit(r) \Rightarrow RET$ combines a QTree method name with input values and a return value $RET \in \{OK, FAIL\}$. An invocation label is called *successful* when the return value is OK. A sequence σ of invocation labels is called *correct* when there exist QTree states $q_0, \ldots, q_{|\sigma|}$, such that q_0 is the QTree initial state and for each $i \in [1, |\sigma|]$, executing σ_i starting from q_{i-1} leads to q_i.

Theorem 8. *A sequence σ of successful invocation labels is correct if and only if the following hold (we use _ to denote arbitrary values):*

1. *for every r, σ contains at most one invocation label $add(r,\_,\_)$ and at most one invocation label $commit(r)$*
2. *every $commit(r)$ is preceded by an $add(r,\_,\_)$*
3. *if $r_p > 0$, every $add(r, v, r_p)$ is preceded by $add(r_p, v',\_)$ where $0 < r_p < r$*
   a) and $v = v'$
4. *if σ contains $add(r,\_,\_)$ and $add(r',\_,r'')$ with $r'' < r < r'$, then σ does* not *contain $commit(r)$*

Properties 1–3 are straightforward consequences of the add and $commit$ definitions. Indeed, it is impossible to add two nodes with the same round number r, which implies that there can not be two successful $add(r, \_, \_)$ invocations, the status of a node can be flipped to `COMMITTED` exactly once, which implies that there can not be two successful $commit(r)$ invocations, and a $commit(r)$ is successful only if a node with round number r already exists, hence Property 2 must hold. Moreover, a node's parent defined by the input r_p must already exist in the tree, which implies that Property 3 must also hold. Property 4 is more involved and relies on the fact that a node n with round number r can be `COMMITTED` only if there exist no other conflicting node n' with a bigger round number r' (the parent of n' having a round smaller than r implies that n and n' are conflicting). This is an inductive invariant of reachable QTree states.

Proof. ($\Rightarrow$): Assume that σ is correct. We show that it satisfies the above properties:

- Property 1: The $newRound(n)$ predicate used at line 8 in Algorithm 4 ensures that it is impossible to add two nodes with the same round number r, and therefore σ can not contain two successful $add(r, \_, \_) \Rightarrow OK$ invocations. The conditions at line 19 ensure that $commit(r) \Rightarrow OK$ can flip the status of a node only once, and therefore only one such successful invocation can occur in σ.

- Property 2: The conditions at line 19 in Algorithm 4 imply that the state in which $commit(r) \Rightarrow OK$ is executed contains a node with round number r. This node could have only added by a previous $add(r, \_, \_) \Rightarrow OK$ invocation.

- Property 3: The $link(n)$ predicate used at line 8 in Algorithm 4 ensures that the state in which $add(r, v, r_p) \Rightarrow OK$ is executed contains a node with round number r_p. This node could have only added by a previous $add(r_p, v', \_) \Rightarrow OK$ invocation, for some v'.

  - Property 3a: It is a direct consequence of the $valueConstraint(n)$ predicate used at line 8 in Algorithm 4.

- Property 4: Let n and n' be the nodes of the QTree state q reached after executing σ, which have been added by $add(r, \_, \_) \Rightarrow OK$ and $add(r', \_, r'') \Rightarrow OK$, respectively. We have that n'.round $>$ n.round $>$ n.parent.round. Since the round numbers decrease when going from one node towards $Root$ in a reachable QTree state, it must be the case that n and n' are conflicting. By Lemma 8, we get that n.status is `GHOST`. Since the `GHOST` status can not be turned to `COMMITTED` and vice-versa, it follows that σ can not contain $commit(r) \Rightarrow OK$.

($\Leftarrow$): We prove that every sequence σ that satisfies properties 1–4 is correct. We proceed by induction on the size of σ. The base step is trivial. For the induction step, let σ be a sequence of size $k+1$. If σ satisfies properties 1-4, then the prefix σ' containing the first k labels of σ satisfies properties 1-4 as well. By the induction hypothesis, σ' is correct. We show that the last invocation of σ, denoted by σ_{k+1} can be executed in the QTree state $q_{|\sigma'|}$ reached after executing σ'. We start with a lemma stating an inductive invariant for reachable QTree states:

Lemma 8. *For every node n in any state q reached after executing a correct sequence σ of successful invocations, n.status is `COMMITTED` if n is $Root$ or σ contains a $commit(r)$ invocation. Else, n.status is `GHOST` if q contains a node n' with n'.round $>$ n.round and n' is conflicting with n, and it is `ADDED`, otherwise.*

Proof. We proceed by induction on the size of σ. The base step is trivial. For the induction step, let σ be a sequence of size $m+1$. Let q_m be the state reached after executing the prefix of size m of σ, and let σ_{m+1} be the last invocation label of σ. We show that the property holds for any possible σ_{m+1} that takes the QTree state q_m to some other state q_{m+1}:

- $\sigma_{m+1} = add(r, v, r_p) \Rightarrow OK$, for some r, v, r_p: Let n be the new node added by this invocation. The status of n can be `ADDED` or `GHOST`. If q_m contains a node n' with n'.round $> r$ (since round numbers are decreasing going towards the $Root$ and n is a new leaf node, any existing node with a higher round number such as n' is also conflicting with n), then the status of n becomes `GHOST` by the predicate at line 11 in Algorithm 4 (otherwise, it remains `ADDED`). This implies that n's status satisfies the statement in the lemma. This invocation may also turn the status of some set of nodes N from `ADDED` to `GHOST` by the statement at line 13 in Algorithm 4. The nodes in N have a lower round number than r and conflicting with n. Therefore, the statement of the lemma is satisfied for the nodes in N.

- $\sigma_{m+1} = commit(r) \Rightarrow OK$, for some r: For $commit(r)$ to be successful the conditions at line 19 in Algorithm 4 must be satisfied. If it is satisfied, only the status of node n is changed from `ADDED` to `COMMITTED`. Note that $Root$ exists by definition and its status is `COMMITTED`. Since the statuses of the rest of the nodes stay the same, the statement of the lemma holds.

□

There are two cases to consider depending on whether σ_{k+1} is an add or $commit$ invocation label:

- $add(r, v, r_p)$: This invocation label is successful if and only if the predicates $valid(n)$ and $valueConstraint(n)$ at line 8 in Algorithm 4 are satisfied after generating a new node n with the given inputs in the state $q_{|\sigma'|}$:
  - $newRound(n)$: Due to Property 1, $r \neq n'$.round for any other node $n' \in q_{|\sigma'|}$ and the predicate is satisfied.
  - $link(n)$: To satisfy this predicate, there must exist a node in $q_{|\sigma'|}$ with round r_p where $r_p < r$. By Property 3, if σ contains $add(r, \_, r_p) \Rightarrow OK$ with $r_p \neq 0$, then $add(r_p, \_, \_) \Rightarrow OK$ also exists in σ. Hence, there exists a node p with round r_p in $q_{|\sigma'|}$, and the predicate is satisfied. If $r_p = 0$, then $q_{|\sigma'|}$ contains the $Root$ node (with round 0) which ensures that the predicate is satisfied.

- $extendsTrunk(n)$: This predicate states that n extends the node n' which has the highest round number among the nodes with `COMMITTED` status, if n.round > n'.round. Assume by contradiction that this is not the case, i.e., n.round > n'.round but n and n' are conflicting. Let n_1 be the lowest common ancestor of n and n' (the first common node on the paths from n and n' to the $Root$). Since the round numbers decrease when going from one node towards $Root$, we have that n_1.round < n'.round. If we consider the nodes on the path from n to n_1, since n.round > n'.round, there must exist a node n_2 such that n_2.round > n'.round but n_2.parent.round < n'.round. The node n_2 in $q_{|\sigma'|}$ corresponds to the invocation label $add(n_2.\text{round},\_, n_2.\text{parent.round})$ in σ'. Moreover, the `COMMITTED` status of n' implies the existence of $commit(n'.\text{round})$ in σ' as stated in Lemma 8. However, it is impossible that σ' contains both these invocation labels if Property 4 holds.
- $valueConstraint(n)$: It is implied trivially as Property 3a holds.

- $commit(r)$: It is successful if and only if the conditions at line 19 in Algorithm 4 are satisfied. Then by Property 1 and 2, there exist $add(r, \_, \_)$ in σ' but not $commit(r)$. As $add(r, \_, \_)$ is successful, there already exist a node n in $q_{|\sigma'|}$ where its round is r but its status can be either `ADDED` or `GHOST`. Towards a contradiction, assume that n.status = `GHOST` in $q_{|\sigma'|}$. This means that there exists a node n' conflicting with n such that n'.round > n.round as stated in Lemma 8. Let n_1 be the least common ancestor of n and n'. Since round numbers are decreasing going towards the $Root$, n_1.round < n.round. If we consider nodes on the path from n' to n_1, there exists a node n_2 such that n_2.round > n.round and n_2.parent.round < n.round. That's why, there is an invocation label $add(n_2.\text{round},\_, n_2.\text{parent.round})$ in σ'. However, σ cannot contain both of these invocation labels together according to Property 4.

□

3.4 Linearization Points

We describe an instrumentation of consensus protocols with linearization points of successful QTree invocations, and illustrate it using Paxos as a running example. Section 3.5–3.8 will discuss other protocols like HotStuff, Raft, PBFT, and Multi-Paxos.

The identification of linearization points relies on the fact that protocol executions pass through a number of rounds, and each round goes through several phases (rounds can run asynchronously – processes need not be in the same round at the same time). The protocol imposes a total order over the phases inside a round and among distinct rounds. Processes executing the protocol can

only move forward following the total order on phases/rounds. Going from one phase to the next phase in the same round is possible if a quorum of processes send a particular type of message. The refinement proofs require identifying two quorums for each round where a value is first proposed to be agreed upon and then decided. They correspond to linearization points of successful $add(r, \_, \_)$ and $commit(r)$, respectively.

The linearization point of $add(r, v, r_p) \Rightarrow OK$ occurs when intuitively, the value v is proposed as a value to agree upon in round r. For the protocols we consider, v is determined by a designated leader after receiving a set of messages from a quorum of processes. For single-decree consensus, members of the quorum send the latest round number and value they adopted (voted) in the past and the leader picks a value corresponding to the maximum round number r_p. If no one in the quorum has adopted any value yet, then the leader is free to propose any value received from a client, and r_p equals a default value 0. For state-machine replication protocols like Raft or HotStuff, the round r_p is defined in a different manner – see Section 3.5 and Section 3.6. The linearization point of $commit(r) \Rightarrow OK$ occurs when a quorum of nodes adopt (vote for) a value v proposed at round r.

By Theorem 8, proving that the order between linearization points along a protocol execution defines a correct QTree execution reduces to showing Properties 1–4. In general, Properties 1–3 are quite straightforward to establish and follow from the control-flow of a process. Property 3a is specific to single-decree consensus protocols or compositions thereof, e.g., (multi-)Paxos and PBFT. It will not hold for Raft or Hotstuff. Property 4 is related to the fact that any two quorums of processes intersect in a correct process.

Above, we have considered the case of a protocol that is a refinement of a single instance of QTree. State machine replication protocols that are composed of multiple independent consensus instances, e.g., PBFT (see Section 3.7), are refinements of a set of QTree instances (identified using a sequence number) and every linearization point needs to be associated with a certain QTree instance.

3.4.1 Linearization Points over Paxos

For concreteness, we exemplify the instrumentation with linearization points on the single-decree Paxos protocol. We start with a brief description of this protocol that focuses on details relevant to this instrumentation.

3.4.1.1 Description of the Protocol

Paxos proceeds in rounds and each round has a unique leader. Since the set of processes running

the protocol is fixed and known by every process, the leader of each round can be determined by an a-priorly fixed deterministic procedure (e.g., the leader is defined as $r \ mod \ N$ where r is the round number and N the number of processes). For each round, the leader acts as a proposer of a value to agree upon.

A round contains two phases. In the first phase, the leader broadcasts a `START` message to all the processes to start the round, executing the **`START`** action below, and processes acknowledge with a `JOIN` message if some conditions are met, executing the **`JOIN`** action:

- **`START` Action:**

  The leader p of round $r > 0$ (the proposer) broadcasts a `START`(r) message to all processes.

- **`JOIN` Action:**

  When a process p' receives a `START`(r) message, if p' has not sent a `JOIN` or `VOTE` message (explained below) for a higher round in the past[1], it replies by sending a `JOIN`(r) message to the proposer. This message includes the maximum round number ($maxVotedRound$) for which p' has sent a `VOTE` message in the past and the value ($maxVotedValue$) proposed in that round. If it has not voted yet, these fields are 0 and $\perp$.

If the leader receives `JOIN` messages from a quorum of processes, i.e., at least $f + 1$ processes from a total number of $2f + 1$, the second phase starts. The leader broadcasts a `PROPOSE` message with a value, executing the **`PROPOSE`** action below. Processes may acknowledge with a `VOTE` message if some conditions are met, executing a **`VOTE`** action. If the leader receives `VOTE` messages from a quorum of processes, then the proposed value becomes decided (and sent to the client) by executing a **`DECIDE`** action:

- **`PROPOSE` Action:**

  When the proposer p receives `JOIN`(r) messages from a quorum of ($f + 1$) processes, it selects the one with the highest vote round number and proposes its value by broadcasting a `PROPOSE`(r) message (which includes that value). If there is no such highest round (all vote rounds are 0), then the proposer selects the proposed value randomly simulating a value given by the client (whose modeling we omit for simplicity).

- **`VOTE` Action:**

  When a process p' receives a `PROPOSE`(r) message, if p' has not sent a `JOIN` or `VOTE` message for a higher round in the past, it replies by sending a `VOTE`(r) message to the proposer with round number r.

[1] Each process has a local variable $maxJoinedRound$ that stores the maximal round it has joined or voted for in the past and checks whether $maxJoinedRound < r$

- **DECIDE Action:**

  When the proposer p receives VOTE(r) messages from a quorum of processes, it updates a local variable called $decidedVal$ to be the value it has proposed in this round r. This assignment means that the value is decided and sent to the client.

3.4.1.2 Linearization points in Paxos

We instrument Paxos with linearization points as follows:

- the linearization point of $add(r, v, r') \Rightarrow OK$ occurs when the proposer broadcasts the PROPOSE(r) message containing value v after receiving a quorum of JOIN(r) messages (during the **PROPOSE** action in round r). The round r' is extracted from the JOIN(r) message selected by the proposer.

- the linearization point of $commit(r) \Rightarrow OK$ occurs when the proposer who is the leader of the round r updates $decidedVal$ with value v after receiving a quorum of VOTE(r) messages (during the **DECIDE** Action in round r).

We illustrate the mapping between linearization points in Paxos and QTree using Paxos execution in Figure 3.2 (Paxos):

- The leader p_1 of round 1 starts the round by broadcasting a START(1) message; p_1 and p_2 reply with JOIN messages containing empty payloads as they have not sent a VOTE yet. Since p_1 receives a quorum of JOIN messages and there is no highest voted round yet, p_1 selects a random value (v_1), and broadcasts PROPOSE(1). In this step, the linearization point of $add(1, v_1, 0)$ occurs and it is simulated in QTree with an invocation that adds node n_1 with status ADDED. Only p_2 replies with a VOTE message, and all the other messages sent in this round are lost.

- The leader p_2 of round 2 initiates the round. Only p_1 and p_2 reply with JOIN messages. Since p_2 has voted in round 1, it sends the round and the value of the vote. The leader p_2 is slow and will resume later.

- The leader p_3 initiates round 3, and only p_1 and p_3 reply with JOIN messages that contain empty payloads (they have not voted yet). Hence, p_3 selects a random value v_2 and broadcasts PROPOSE(3). Here, the linearization point of $add(3, v_2, 0)$ occurs and it is simulated with a QTree invocation that results in changing the status of n_1 to GHOST and adding a new node n_3 with status ADDED.

- Process p_2 resumes and since it received a quorum of JOIN messages in round 2, p_2 broadcasts PROPOSE(2) by selecting v_1 as the value of the highest voted round from the JOIN messages. At this point, the linearization point of $add(2, v_1, 1)$ occurs and it is simulated by the QTree invocation that adds node n_2 with status **GHOST**.

- Processes p_1 and p_3 continue by voting for the proposal in round 3. As a quorum of VOTE messages is received by p_3, it decides on v_2. Now, the linearization point of $commit(3)$ occurs which is simulated by changing the status of n_3 to **COMMITTED** in QTree.

Our main correctness theorem for Paxos is as follows:

Theorem 9. *Paxos refines QTree.*

Proof. We show that the sequence of successful add and $commit$ invocations defined by linearization points along a Paxos execution satisfies the properties in Theorem 8 and therefore, it represents a correct QTree execution:

- **Property 1:** Each round has a unique leader and the leader follows the rules of the protocol (no Byzantine failures), thereby, making a single proposal. Therefore, the linearization point of an $add(r, \_, \_) \Rightarrow OK$ will occur at most once for a round r. Since a single value can be proposed in a round, and all processes follow the rules of the protocol, they can only vote for that single value. Thus, at most one linearization point of $commit(r) \Rightarrow OK$ can occur for a round r.

- **Property 2:** This holds trivially as all the processes follow the rules of the protocol and they need to receive a PROPOSE(r) message (which can occur only after the linearization point of an $add(r, \_, \_) \Rightarrow OK$) from the leader of round r to send a VOTE(r) message.

- **Property 3:** By the definition of the **PROPOSE** action, the proposer selects a highest vote round number r' from a quorum of JOIN(r) messages that it receives, before broadcasting a PROPOSE(r) message. If such a highest vote round number $r' > 0$ exists, then there must be a VOTE(r') message which is a reply to a PROPOSE(r') message. Thus, if the linearization point of $add(r, \_, r') \Rightarrow OK$ occurs where $r' \neq 0$, then it is preceded by $add(r', \_, \_)$. Also, by the definition of **JOIN**, a process can not send a JOIN(r) message after a VOTE(r') message if $r \not> r'$.

  - **Property 3a:** By the definition of **PROPOSE**, the proposer selects the JOIN message with the highest vote round number and proposes its value. Thus, if the linearization points of both $add(r, v, r') \Rightarrow OK$ and $add(r', v', \_) \Rightarrow OK$ occur, then $v = v'$.

- **Property 4:** Assume by contradiction that the linearization point of $commit(r) \Rightarrow OK$ occurs along with the linearization points of $add(r, \_, \_) \Rightarrow OK$ and $add(r', \_, r'') \Rightarrow OK$,

for some $r'' < r < r'$. The linearization point of $commit(r)$ occurs because of a quorum of VOTE(r) messages sent by a set of processes P_1, and $add(r', \_, r'')$ occurs because of a quorum of JOIN(r') messages sent by a set of processes P_2. Since P_1 and P_2 must have a non-empty intersection, by the definition of **JOIN**, it must be the case that $r'' \geq r$, which contradicts the hypothesis. □

The proof of Property 4 relies exclusively on the quorum of processes in the first phase of a round intersecting the quorum of processes in the second phase of a round. It is not needed that quorums in first, resp., second, phases of different rounds intersect. This observation is at the basis of an optimization that applies to non-Byzantine protocols like Flexible Paxos [53] or Raft (see Section 3.8 and Section 3.6, respectively).

3.4.2 Inferring Safety

We exhibited that if a protocol is shown to be a refinement of QTree (or a composition of independent QTree instances), i.e., its executions are mapped to correct QTree executions (satisfying the properties in Theorem 8), then it satisfies agreement and validity. We showed that for a protocol, there can be only one linearization point of successful add invocation and one linearization point of successful $commit$ invocation for a certain round r. This is because there can be only one proposal of a protocol that can take approval from a quorum of processes and this proposal on a value v in a round r corresponds to linearization point of successful $add(r, v, r')$. If $r' \neq 0$ and therefore, $v' \neq \perp$, then there exist another proposal on a value v' in a round $r' < r$ with the same properties, which is selected by the leader of the round r to propose the same value or connect it with the proposal of round r when commands (values) lie on a log or a branch. This implies that corresponding QTree state contains a node n with n.round = r, n.value = v, and n.status = ADDED which also contains another node n' with n'.round = r', n'.value = v'. Similarly, a decision on a value v in a round r of a protocol corresponds to linearization point of successful $commit$ invocation, which implies that the QTree state contains a node n with n.round = r, n.value = v, and n.status = COMMITTED.

For single-decree consensus (or multi-decree consensus protocols designed as multiple instances of single-decree consensus protocols), Proposition 5 ensures that there is only one value which is decided and it is guaranteed by $valueConstraint(n)$ and $extendsTrunk(n)$ predicates. For state machine replication protocols like Raft and HotStuff, where the goal is to agree on a sequence of commands, Proposition 4 ensures that all the decided values lie on the same branch of the tree since the values propagated between different rounds by the selection of leaders is preserved by the properties of linearization points, which concludes that all processes agree on the same sequence of commands.

For validity, when $valueConstraint(n)$ is considered, successful $add(r, v, 0)$ invocations represent proposals of client values since the quorums of acceptors that joined this round have not voted yet. Theorem 8 ensures that these invocations correspond to nodes n that are immediate children of $Root$ and for any such node n, n.value = v. Therefore, by Proposition 3, we can conclude that only client values can be decided. When $valueConstraint(n)$ is not considered, the fact that the value of each node is obtained from a client is ensured using additional mechanisms that are straightforward, e.g., a client broadcasting a command to all the participants in the protocol.

3.5 HotStuff Refines QTree

We present an instrumentation of HotStuff with linearization points of successful add and $commit$ invocations. We use HotStuff as an example of a state machine replication protocol where processes agree over a sequence of commands to execute, and any new command proposed by a leader to the other processes comes with a well-identified immediate predecessor in this sequence. Agreement over a command entails agreement over all its predecessors in the sequence. This is different from protocols such as PBFT or Multi-Paxos, discussed in Section 3.7 and Section 3.8, respectively, where commands are associated to indices in the sequence and they can be agreed upon in any order. Next section presents an instrumentation of Raft which behaves in a similar manner.

3.5.1 Description of the Protocol

In HotStuff, f out of a total of $N = 3f + 1$ processes might be Byzantine in the sense that they might show arbitrary behavior and send corrupt or spurious messages. However, they are limited by cryptographic protocols. HotStuff requires that messages are signed using public-key cryptography, which implies that Byzantine processes cannot imitate messages of correct (non-faulty) processes. Additionally, after receiving a quorum of messages, leaders must include certificates in their own messages to prove that a quorum has been reached. These certificates are constructed using threshold signature schemes and correct processes will not accept any message from the leader if it is not certified. Because of Byzantine processes, HotStuff requires quorums of size of $2f + 1$ which ensures that the intersection of any two quorums contains at least one correct process.

Each process stores a tree of commands. When a node in this tree (representing some command) is decided, all the ancestors of this node in the tree (nodes on the same branch) are also decided. For a node to become decided, a leader must receive a quorum of messages in 3 consecutive phases after the proposal. After each quorum is established, the leader broadcasts a different certificate to state which quorum has been achieved and the processes update different local variables

accordingly, with the same node (if the certificate is valid). These local variables are $preNode$, $votedNode$ and $decidedNode$ in the order of quorums.

To start a new round, processes send their $preNode$'s to the leader of the next round in ROUND−CHANGE(r) messages and increment their round number. After getting a quorum of messages and selecting the $preNode$ with the highest round, the leader broadcasts a PROPOSE(r) message with a new node (value is taken from the client) whose parent is the selected $preNode$. When the message is received by a process, it first checks if the new node extends the selected $preNode$. Then it accepts the new node if the node extends its own $votedNode$ (it is a descendant of $votedNode$ in the tree) or it has a higher round number than the round number of its $votedNode$, and sends[1] a JOIN(r) message with the same content. In the second (resp., third) phase, if a quorum of JOIN(r) (resp., PRECOMMIT_VOTE(r)) messages is received by the leader, it broadcasts a PRECOMMIT(r) (resp., COMMIT(r)) message, and processes update their $preNode$ (resp., $votedNode$) with the new node, sending a PRECOMMIT_VOTE(r) (resp., COMMIT_VOTE(r)) message. In the fourth phase, when the leader receives a quorum of COMMIT_VOTE(r), it broadcasts a DECIDE(r) message and processes update their $decided\text{-}Node$ accordingly. You can find for more detailed description below:

In the first phase, the leader broadcasts a PROPOSE message to all the processes with a node by executing the **PROPOSE** action below. This nodes value is sent by a client whose modeling we omit for simplicity. Parent of this node is obtained from processes' ROUND−CHANGE messages. Processes acknowledge with a JOIN message if some conditions are met, executing a **JOIN** action below:

- **PROPOSE Action:**

  When a proposer p who is the leader of the new round r, receives a quorum ($2f + 1$) of ROUND−CHANGE(r) with $preNode$, it selects the node with the highest round from this set of $preNode$'s. Then it extends the selected node by a newly created node which is initialized with the current round and some value. If there is no such highest round, then the proposer extends the Genesis Block. Finally, proposer p broadcasts PROPOSE(r) to all processes alongside with the new node.

- **JOIN Action:**

  When a process p' receives a PROPOSE(r) with node n, if n extends $votedNode$ or the round of the $votedNode$ is less than the round of $n.parent$, then p' sends a JOIN(r) to the leader of the current round.

[1] For all received messages, a correct process also checks if the round number of the node sent by the leader is equal to the current round number of its own, and can send only one message for each phase in each round.

If the leader receives acknowledgement messages from a quorum of processes, the second phase starts. The leader broadcasts a PRECOMMIT message with the same node by executing **PRE-COMMIT** action and accepters acknowledge with the PRECOMMIT_VOTE message, executing **PRECOMMIT_VOTE** action.

- **PRECOMMIT Action:**

  When the proposer p receives a quorum of JOIN(r) with its current round and the same node, p combines (generates certificate) and sends them by broadcasting a PRECOMMIT(r) with the same node n to all processes.

- **PRECOMMIT_VOTE Action:**

  When a process p' receives a PRECOMMIT(r) from the leader of its current round, p' updates $preNode$ with the node n that it received and sends a PRECOMMIT_VOTE(r) with the same node to the leader of the current round.

Like the previous phase, if the leader receives acknowledgement messages from a quorum of processes, the third phase starts. The leader broadcasts a COMMIT message with the same node by executing **COMMIT** action and accepters acknowledge with the COMMIT_VOTE message, executing **COMMIT_VOTE** action. Then if the leader receives COMMIT_VOTE messages from a quorum of processes, the proposed value becomes decided (and sent to the client) by executing a **DECIDE** action.

- **COMMIT Action:**

  When the proposer p receives a quorum of PRECOMMIT_VOTE(r) with its current round and the same node, p combines and sends them by broadcasting a COMMIT(r) with the same node n to all processes.

- **COMMIT_VOTE Action:**

  When a process p' receives a COMMIT(r) from the leader of its current round, p' updates $votedNode$ with the node n that it received and sends a COMMIT_VOTE(r) with the same node to the leader of the current round.

- **DECIDE Action:**

  When the proposer p receives a quorum of COMMIT_VOTE(r) with its current round and the same node, p combines and sends them by broadcasting a DECIDE(r) with the same node n to all processes. When a process p' receives a DECIDE(r) from the leader of its current round, p' updates $decidedNode$ as n and execute commands through the branch where the leaf node is n.

If timeout is reached for a process, **ROUND-CHANGE** action will be executed.

- **ROUND−CHANGE Action:**

  When the timeout is reached, a process p' sends a ROUND−CAHNGE(r) with $preNode$ to the leader of the next round. Additionally, p' increments its round number.

3.5.2 Linearization Points in HotStuff

For HotStuff, the linearization points of *add* and *commit* occur with the broadcasts of PRECOMMIT(r) and DECIDE(r) messages, respectively, that are *valid* , i.e., (1) they contain certificates for quorums of JOIN(r) or COMMIT_VOTE(r) messages, respectively, which respect the threshold signature scheme, and (2) they contain the same node as in those messages. More precisely,

- the linearization point of $add(r, v, r') \Rightarrow OK$ occurs the *first* time when a *valid* PRECOMMIT(r) message containing node v is sent. r' is the round of the node which is the parent of v and it is contained in a previous PROPOSE(r) message (r' can be 0 in which case parent of v is a distinguished root node that exists in the initial state).
- the linearization point of $commit(r) \Rightarrow OK$ occurs the *first* time when a *valid* DECIDE (r) message is sent.

Note that a Byzantine leader can send multiple *valid* PRECOMMIT(r) messages that include certificates for different quorums of JOIN(r) messages. A linearization point occurs when the first such message is sent. Even if processes reply to another valid PRECOMMIT(r) message sent later, this later PRECOMMIT(r) message contains the same $preNode$ value, and their reply will have the same content. The same holds for DECIDE(r) messages. This remark along with the restriction to valid messages and the fact that any two quorums intersect in at least one correct process implies that the sequence of successful *add* and *commit* invocations defined by these linearization points satisfies the properties in Theorem 8 and therefore,

Theorem 10. *HotStuff refines QTree.*

Proof. We show that the sequence of successful *add* and *commit* invocations defined by linearization points along a HotStuff execution satisfies the properties in Theorem 8 and therefore, it represents a correct QTree execution:

- **Property 1:** To generate a valid (certified under threshold signatures) PRECOMMIT(r) (resp., DECIDE(r)), a leader must collect a quorum of JOIN(r) (resp., COMMIT_VOTE(r)) messages with the same content i.e., the same node with the same client request, connected to the same parent. As all the correct replicas will send at most one message per phase in a single round r, there can't be two quorums of JOIN(r) (resp., COMMIT_VOTE(r))

resulting two PRECOMMIT(r) with different contents. Since the linearization point of $add(r, \_,)$ (resp., $commit(r)$) occurs when the $first$ valid PRECOMMIT(r) (resp., DECIDE(r)) message is broadcasted, property holds by the definition.

- **Property 2:** This holds trivially as there won't be a quorum of COMMIT_VOTE(r) messages without a quorum of PRECOMMIT_VOTE(r) messages which do not exist as there is no valid PRECOMMIT(r).

- **Property 3:** By the definition of **JOIN** action, a correct process will accept a PROPOSE(r) message if the node that is sent alongside with this message is extending some $preNode$ which can be certified only if a quorum of JOIN(r') messages (that forms a PRECOMMIT(r')) are sent to the leader for some round $r' > 0$. Since a quorum of JOIN(r') messages and PROPOSE(r) message are formed before and after a quorum of ROUND-CHANGE(r) respectively, $r > r'$. Note that processes can only vote for their current round and the round number monotonically increases. Therefore, to reach a quorum of JOIN(r) (which is imperative to generate PRECOMMIT(r)), PRECOMMIT(r') must exists.

  - **Property 3a:** This property doesn't hold (and not needed) for HotStuff.

- **Property 4:** Assume by contradiction that $commit(r)$ occurred along with the other two linearization points of add. Linearization point of $commit(r)$ exists because of a quorum of COMMIT_VOTE(r) messages sent by a set of processes P_1, and $add(r', \_, r'')$ exists because of a quorum of JOIN(r') messages sent by a set of processes P_2. All the correct processes in P_1 must updated their $votedNode$ with a node whose round is r when they sent COMMIT_VOTE(r) message. But also, all the correct processes in P_2 must have a $votedNode$ whose round number is less than or equal to r'' by the predicates in the definition of the **JOIN(r)** action. Note that none of the correct processes in P_2 can send COMMIT_VOTE(r) message anymore since their current round number is at least r' which is greater than r. Since P_1 and P_2 must have an intersecting correct process, it contradicts the hypothesis as $r'' < r$.

□

3.6 Raft Refines QTree

3.6.1 Description of the Protocol

Raft is a partially synchronous, multi-decree consensus protocol that is resilient to only crash-restart failures. Each process of Raft keeps a durable log for storing the sequence of commands.

Raft has the notion of *terms* that does not exactly match with our round notion. For each term, there is at most one leader that does not change throughout the term. When the current term's leader is suspected to fail, processes start a new leader election for a new bigger term number. The leader might propose values for different commands (log indices) inside a term as long as it stays alive. In order to differentiate values proposed and decided for distinct commands within a term, we keep term-index pairs ($r = (t, idx)$) as rounds. We assume the usual lexicographical total ordering on rounds.

Logs keep term-value pair at each index. Here, the first field represents the term at which this element is created and inserted to the log and the second field represents the command offered for this index. For each process, a prefix of the log is called decided. If it is decided, then this prefix is supposed to be the same for a majority of process logs. Remaining parts of the logs (suffixes) might be different among processes. Length of the uncommon suffix might be more than one and different between processes since a leader might append multiple items to the log at once and these new entries might arrive to a subset of processes. For each process, we keep two special indices: $didx_p$ marks the end of the decided prefix whereas $lidx_p$ shows the last entry's index for the log of process p.

Raft rounds consist of a single main phase ignoring the leader election phase that does not happen at every round. Leader election phases are only executed during term changes. The leader election phase can be considered as the first phase of the first round of the new term.

When a process p suspects from the inactivity of the current leader, it broadcasts `VOTEREQ` message to initiate the leader election phase and it becomes the candidate leader for the new term. When a process p' receives this message, it responds to p with a `VOTERESP` message if some conditions are met. If p can get `VOTERESP` messages from a majority, it becomes the leader of the new term.

- **`VOTEREQ` Action:**

  This action is executed when process p times out while waiting for a message from the leader of the term $t - 1$. Process p broadcasts `VOTEREQ`$(t, lidx_p)$ message alongside with $log_p[lidx_p]$ and updates its term $term_p$ to t.

- **`VOTERESP` Action:** This action can only be executed by process p' after receiving some `VOTEREQ`$(t, lidx_p)$ action from the candidate leader p. With this action, p' sends the message `VOTERESP`$(t, lidx_{p'})$ to p and updates its term $term_{p'}$ to t if (1) it has not sent any `VOTERESP` message for the term t or higher before ($t \geq term_{p'}$), and (2) $(log_{p'}[lidx_{p'}].term, lidx_{p'}) \leq (log_p[lidx_p].term, lidx_p)$. Second condition means that the last item in p's log has been proposed in a bigger or the same round than the last item in p''s log.

If p can collect VOTERESP messages from a majority, p becomes the leader of term t and start proposing values. Since a process can send at most one VOTERESP message for any term, there is at most one leader for each term.

After the leader is elected, it starts sending requests to processes to append new values to their logs by iterating the main phase of rounds. When new commands come from clients, the leader p first appends them to its own log with the current term, increments $lidx_p$ and then broadcasts LOGREQ messages that include the new entries. When a process p' receives this message, it checks some conditions. If conditions are satisfied, it updates its $didx_{p'}$, $lidx_{p'}$ and $log_{p'}$ and then responds with a LOGRESP message to the leader. If the leader receives a LOGRESP message from a majority, it confirms that the new commands became permanent in a majority of processes and updates its $didx_p$ value.

- **LOGREQ Action:**

  This action is executed by the leader process p of $term_p$. If this action is not the first **LOGREQ** action of this term, the leader first checks if there is a set of LOGRESP messages from a majority for the previous **LOGREQ** action. If this is the case, it updates $didx_p$ value to $lidx_p$ and decides on the entries appended in the previous turn. Then, it appends new entries to log_p and updates $lidx_p$ so that it now points to the end of log_p. As the last thing, it broadcasts LOGREQ$(t, lidx_p)$ message with its log_p and $didx_p$.

- **LOGRESP Action:**

  This action can be only executed by process p' after receiving a LOGREQ message. First, p' checks whether $term_p \geq term_{p'}$ and $lidx_p \geq lidx_{p'}$. If this is the case, it updates $term_{p'}$, $lidx_{p'}$ and $didx_{p'}$ to $term_p$, $lidx_p$ and $didx_p$, respectively. Moreover, for each index $i : 0 \leq i \leq lidx_p$, it replaces $log_{p'}[i]$ with $log_p[i]$. Then it sends LOGRESP$(t, lidx_{p'})$ response back to the leader process p.

Even if the leader p does not receive a new value from the clients for a long time, it still broadcasts a LOGREQ message with the same log_p and $lidx_p$ value as the previous LOGREQ message to signal to other processes that it is still alive. These LOGREQ messages are called *heartbeat* messages. They can only differ on $didx_p$ values since a majority quorum might send LOGRESP messages in between two heartbeat messages that can change the $didx_p$ values. We also include heartbeat messages in our formulation.

3.6.2 Linearization Points in Raft

Inside a term t, if the leader p receives a LOGRESP message from a majority, this quorum becomes a witness for the decision of entries in the log and proposal of the new entries that will be

coming from the clients. In some sense, they correspond to VOTE and JOIN quorums of Paxos, respectively. When the leader broadcasts a LOGREQ message first time in a new term, the witness quorum for the proposal of new entries is formed by VOTERESP messages received from a majority. Therefore, both add and $commit$ linearization points correspond to the **LOGREQ** actions.

Assume that old_didx_p to represent the $didx_p$ value before executing the **LOGREQ** action. Then,

- the linearization point of $add((log_p[i].term, i), log_p[i].value, (log_p[i-1].term, i-1)) \Rightarrow OK$ occurs in **LOGREQ** action for all indices i such that $lidx_p \geq i > didx_p$ and $log_p[i].term = term_p$. For the case $i = 0$, we replace the round $(log_p[i-1].term, i-1)$ with $\perp$.

- the linearization point of $commit((log_p[i].term, i)) \Rightarrow OK$ occurs for (again during **LOGREQ** action) for all indices i such that $didx_p \geq i > old\_didx_p$ and $log_p[i].term = term_p$. Note that old_didx_p is not defined if **LOGREQ** is the first such action of $term_p$. Indeed, this action is not a $commit$ linearization point in this case.

Theorem 11. *Raft refines QTree.*

Proof. We show that the sequence of successful add and $commit$ invocations defined by linearization points along a Raft execution satisfies the properties in Theorem 8 but before explaining that, we introduce two additional properties of Raft that will be used during the proofs.

(a) Consider a log of a process in a reachable Raft state. For any two indices $i \leq i' \leq lidx$, we have $log[i].term \leq log[i'].term$.

(b) Assume that $commit(r)$ action is generated for some $r = (t, i)$. Now, consider the log of a leader p for some term $t' \geq t$ during this term t'. We have $log_p[i].term = t$.

Raft's leader election mechanism ensures that there is a unique leader that can execute **LOGREQ** action and append entries to logs at any time and the term of the current leader is bigger than all previously active terms. These ensure property (a).

During the leader election process, processes with the longest logs are the ones that received the latest updates from the previous leader. Since LOGRESP and VOTERESP quorums intersect, if there is a decided entry in one of the previous terms, the new leader has the same entry in its log as well. This ensures property (b).

The proof of the properties for the linearization points along a Raft execution is as the following:

- **Property 1:** Consider any round $r = (t, i)$. Leader election phase of Raft ensures that a unique leader can execute **LOGREQ** actions inside the term t. Moreover, log of the leader only grows and items in previously entered indices never change through a term. Therefore, there is a unique $add(r, \_, \_)$ linearization point for each round r. There is a unique $commit(r)$ linearization point due to previously mentioned properties of the leader and $didx$ of the leader is non-decreasing through a term.

- **Property 2:** Consider a $commit(r)$ linearization point for some $r = (t, i)$. Since $commit(r)$ linearization action exists, **LOGREQ** action a that leads to this point is not the first **LOGREQ** action of this term. Moreover, the last non-heartbeat **LOGREQ** action before this one is the linearization point for $add(r, \_, \_)$.

- **Property 3:** Consider an $add(r, v, r')$ linearization point where $r = (log[i].term, i)$ and $r' = (log[i-1].term, i-1)$ for some $i > 0$. Property (a) ensures that $log[i].term \geq log[i-1].term$. Therefore $r > r'$ according to the lexicographical ordering we have on rounds. Moreover, since there is an item in $log[i-1]$, there must be a **LOGREQ** action that caused this item to be inserted into a log first time. This action must have led to $add(r, v, r')$ linearization point.

  - **Property 3a:** This property does not hold for Raft.

- **Property 4:** Towards a contradiction, assume that there are $add(r, \_, \_)$, $add(r', \_, r'')$ and $commit(r, \_)$ linearization points where $r = (t, i)$, $r' = (t', i')$ and $r'' = (t'', i'')$. Total ordering on rounds ensures that $t' \geq t \geq t''$. In terms of the second fields, the second add linearization point ensures that $i' = i'' + 1$.

  Consider the log of the leader of the term t' that generates the second add at the state it generated this linearization point. Since there is a $commit(r, \_)$ linearization point, property (b) ensures that $log[i].term = t$. Moreover, we have $log[i'].term = t'$ and $log[i''] = log[i' - 1].term = t''$. Next, we consider different cases on i.

  First of all $i = i'$ cannot be true. If this was the case, since $t = log[i].term = log[i'].term = t'$, we would have $r = r'$. Therefore, we consider $i > i'$ as the first case. For this case, we have $t' > t \geq t''$. This case violates property (a) since $i > i'$ but $term[i] = t < t' = term[i']$.

  The last case we consider is $i < i'$. For this case, we have $t' \geq t > t''$. But, again property (a) is violated since $i \leq i''$ but $term[i] = t > t'' = term[i'']$.

□

If we look at the proofs of our properties, the restriction we have on quorums is that LOGRESP and VOTERESP quorums intersect. We do not need two LOGRESP or two VOTERESP quorums to intersect. Therefore, different correct Raft variants can be developed with different quorum sizes. For instance, if the leaders are mostly stable and the network is reliable, one can modify the LOGRESP quorum size to a smaller value so that entries can be appended more efficiently, but VOTERESP quorum size must be increased by the same amount to still enforce safety guarantees.

3.7 PBFT Refines QTree

The protocols discussed above are refinements of a *single* instance of QTree. State-machine replication protocols based Multi-decree consensus like Multi-Paxos or PBFT can be seen as compositions of a number of single-decree consensus instances that run concurrently, one for each index in a sequence of commands to agree upon, and they are refinements of a set of *independent* QTree instances.

3.7.1 Description of the Protocol

PBFT is a multi-decree consensus protocol in which processes aim to agree on a sequence of values. As in HotStuff, f out of a total number of $3f+1$ processes might be Byzantine and quorums are of size at least $2f+1$. To ensure authentication, messages are signed using public-key cryptography. Messages sent after receiving a quorum of messages in a previous phase include that set of messages as a certificate.

A new round r starts with the leader receiving a quorum of ROUND−CHANGE(r) messages (like in HotStuff). Each such message from a process p includes the VOTE message with the highest round (similarly to the **JOIN** action of Paxos) that p sent in the past, for each sequence number that is not yet agreed by a quorum. For an arbitrary set of sequence numbers sn, the leader selects the VOTE message with the highest round and broadcasts a PROPOSE(r,sn) message that includes the same value as in the VOTE message or a value received from a client if there is no such highest round. As mentioned above, this message also includes the VOTE messages that the leader received as a certificate for the selection. When a process receives a PROPOSE(r,sn) message, if r equals its current round, the process did not already acknowledge a PROPOSE(r,sn) message, and the value proposed in this message is selected correctly w.r.t. the certificate, then it broadcasts a JOIN(r,sn) message with the same content (this is sent to all processes not just the leader). If a quorum of JOIN(r,sn) messages is received by a process, then it broadcasts a VOTE(r,sn) message with the same content. If a process receives a quorum of VOTE(r,sn) messages, then the value in this message is decided for sn. When a process sends its highest round number VOTE messages to the leader of the next round (in ROUND−CHANGE messages), it also includes the quorum of

JOIN messages that it received before sending the VOTE, as a certificate. You can find the detailed description below:

In the first phase, the leader broadcasts a PROPOSE message to all the processes with a value by executing the **PROPOSE** action below. This value can be a value sent by a client (whose modeling we omit for simplicity) or it can be obtained from processes' ROUND−CHANGE messages that started this round. Processes accept this proposal with broadcasting a JOIN message if some conditions are met, executing a **JOIN** action below:

- **PROPOSE Action:**

  When a proposer p who is the leader of the new round r, receives a quorum of ROUND−CHANGE(r) with certificates, it selects the valid (contains quorum of matching JOIN alongside) VOTE with the highest round for some available sequence number sn and propose its value by broadcasting a new PROPOSE(r, sn) for the current round r. If there is no such highest round, then p selects the proposed value randomly simulating the value coming from the client. The proposer p also sends the set of VOTE messages included in the ROUND−CHANGE(r) messages it received, to prove that it selected a valid VOTE with the highest round.

- **JOIN Action:**

  When a processor p' receives some number of PROPOSE with its current round, it can only act to one of them if their sequence numbers are the same. During a round, if p' didn't see any PROPOSE with the same sequence number, then it checks whether the proposers selection is correct. After validating that proposer p selected the VOTE with the highest round, p' broadcasts JOIN(r, sn) to all processes using the same content.

If a process receives a quorum of JOIN messages, the second phase starts. The processes broadcast a VOTE message, executing **VOTE** action. Then if a process receives a quorum of VOTE messages, the proposed value becomes decided for this process (and sent to the client) during the execution of **DECIDE** action.

- **VOTE Action:**

  When a process p' receives a PROPOSE and a quorum of JOIN with its current round and the same sequence number, it broadcasts a VOTE(r, sn) to all processes using the same values.

- **DECIDE Action:**

  When a process p' receives a PROPOSE and quorums of JOIN and VOTE with its current round and the same sequence number, it updates a local variable $decidedVal[sn]$ with the value that it received for the order. This assignment means that the value is decided for the

order sn and sent to the client. Since there may be f Byzantine processes, the client accept the decision if it receives $f + 1$ of the same decided value for the same sequence number.

If timeout is reached for a process, **ROUND−CHANGE** action will be executed.

- **ROUND−CHANGE Action:**

  When the timeout is reached, a process p' sends a ROUND−CAHNGE(r) to the proposer of the next round. Additionally, for all the sequence numbers which are not decided by a quorum yet, if p' sent VOTE for some of these sequence numbers before, then p' sends the one with the highest round for each of these sequence numbers. The process p' sends them as a certificate which consists of VOTE, the matching PROPOSE and the quorum of JOIN that it received. Finally, the process p' increments its round number.

3.7.2 Linearization Points in PBFT

PBFT is a refinement of a *set* of independent QTree instances, one for each sequence number. The linearization points will refer to a specific instance identified using a sequence number, e.g., $sn.add(r, v, r')$ denotes an $add(r, v, r')$ invocation on the QTree instance sn. Therefore,

- the linearization point of $sn.add(r, v, r') \Rightarrow OK$ occurs the *first* time when a process p sends a VOTE(r, sn) message, assuming that p is "honest", i.e., it already received a quorum of JOIN(r, sn) messages with the same content. v is the value of the VOTE(r', sn) message that is included in the PROPOSE(r,sn) message (it is possible that $r' = 0$ and v is selected randomly).

- the linearization point of $sn.commit(r) \Rightarrow OK$ occurs the *first* time when a process p decides value v for sn, assuming that p is "honest", i.e., it already received a quorum of JOIN(r, sn), resp., VOTE(r, sn), messages with the same content.

We illustrate the mapping between protocol steps and QTree steps using the PBFT execution for sequence number 1 in Figure 3.3 (PBFT). The corresponding QTree execution is given just above. Therefore:

- The leader p_1 of round 1 starts the round by broadcasting a PROPOSE(1) message where v_1 is selected randomly; p_1, p_2 and p_3 acknowledge the proposal by broadcasting JOIN messages. Now only p_2 broadcasts VOTE$(1, 1, v_1)$ message after recieveing a quorum of JOIN. In this step, the linearization point of $1.add(1, v_1, 0)$ occurs and it is simulated in QTree with an invocation that adds node n_1 with status ADDED.

- When p_1, p_2 and p_4 becomes active, they send ROUND−CHANGE messages to the leader of round 2 which is p_2. Here, since p_2 is already voted for this sequence number, it sends its vote

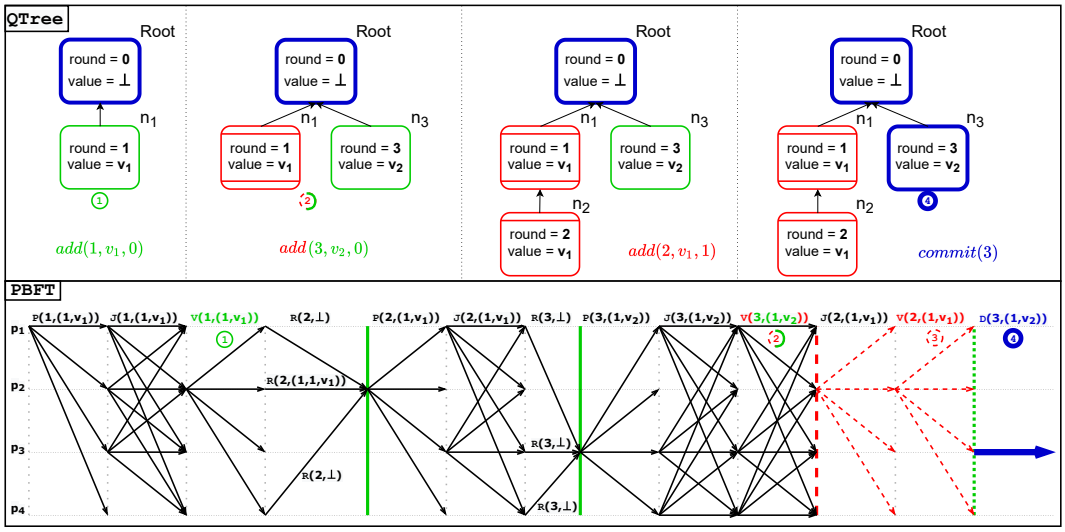

Figure 3.3: **Top:** QTree - Explaining the behavior of *add* and *commit* methods. Colors should be interpreted as in Figure 3.1
Bottom: PBFT - A PBFT execution (for sequence number 1) simulated by the QTree execution above. We abbreviate types of messages with their first letter, e.g., ROUND−CHANGE with R, and payloads are given in parenthesis. Concerning the vertical lines, colors should be interpreted as in Figure 3.2

.

alongside ROUND−CHANGE, others send empty payloads. Then p_2 starts round 2 with sending PROPOSE by selecting v_1 as the value of the highest voted round. But due to some connection loss, no more progress can be done in round 2 as the next quorum is not achieved (there is 1 missing message).

- After that p_2 crashes and so, p_1, p_3 and p_4 send ROUND−CHANGE messages with empty payloads as they haven't voted yet. Hence, p_3 (which is the leader of the round 3) selects a random value v_2 and broadcasts PROPOSE message for round 3. Later in the same round, p_1, p_3 and p_4 continue by sending JOIN and VOTE messages to the leader of the round. Here, the linearization point of $1.add(3, v_2, 0)$ occurs and it is simulated with a QTree invocation that results in changing the status of n_1 to GHOST and adding a new node n_3 with status ADDED

- Process p_2 becomes active again and since it receives a quorum of JOIN messages in round 2 by sending one final message to itself (which is the current round of p_2), p_2 broadcasts VOTE message. At this point, the linearization point of $1.add(2, v_1, 1)$ occurs and it is simulated by the QTree invocation that adds node n_2 with status GHOST.

- Finally, as a quorum of VOTE messages is received, p_3 decides on v_2 and this changes the status of n_3 to COMMITTED due to the occurrence of the linearization point of $commit(3)$.

Our main correctness theorem for PBFT is as follows:

Theorem 12. *PBFT refines a composition of independent QTree instances.*

Proof. A protocol refines a set of QTree instances identified using sequence numbers when it satisfies Properties 1-4 in Theorem 8 for each sequence number, e.g., Property 1 becomes for every sn and every r, a protocol execution contains a linearization point for at most one invocation $sn.add(r, \_, \_) \Rightarrow OK$ and at most one invocation $sn.commit(r) \Rightarrow OK$.

We show that the sequence of successful add and $commit$ invocations on a QTree instance sn defined by linearization points along a PBFT execution satisfies the properties in Theorem 8 and therefore, it represents a correct QTree execution:

- **Property 1:** By the definition of **`JOIN`** action, a correct process can only send a `JOIN`(r, sn) message if this process has not sent a `JOIN` message for the same round and sequence number yet. Hence, there can be at most 1 quorum (2f+1) of `JOIN` messages with the same sequence number and round since any two quorums must intersect with a correct process. This implies that the linearization point of $sn.add(r, \_, \_)$ will occur at most once for a round r with the same sequence number sn. Therefore, a correct process can only vote for a single propose in a round and only 1 proposal can be decided each round. Thus, at most one linearization point of $sn.commit(r)$ can occur for a round r with the same sequence number sn.

- **Property 2:** This holds trivially as there won't be a quorum of `VOTE`(r, sn) messages without a `VOTE`(r, sn) from an honest process since there is no quorum of `JOIN`(r, sn) messages.

- **Property 3:** By the definition of **`PROPOSE`** action, proposer selects a highest vote round number r' from a quorum of `ROUND-CHANGE`(r) messages that it receives for each sequence number, before broadcasting a `PROPOSE`(r, sn) message. Since all processes also send the quorum of `JOIN` messages that they receive as a certificate for each `VOTE` that they send in their `ROUND-CHANGE` messages, a correct proposer can validate `VOTE` messages. If the proposer is faulty, it may select a non-existing vote but correct processes do not proceed with the current proposal since it is not valid. If correct processes can progress with the current proposal where there is a highest vote round number $r' > 0$ selected by the proposer, and one of the processes sends a `VOTE`(r, sn), then there must be a `VOTE`(r', sn) message. Hence, if the linearization point of $sn.add(r, \_, r')$ occurs in PBFT where $r' \neq 0$, then it is preceded by $sn.add(r', \_, \_)$.

  - **Property 3a:** If the proposer is correct, it must propose the same value with the vote that is selected from a quorum of `ROUND-CHANGE` messages by definition. Else if

the proposer omit the definition and send a different value, correct processes do not accept the current proposal and the quorum will not be formed.Therefore, when the linearization point of both $sn.add(r, v, r')$ and $sn.add(r', v', \_)$ occur in a PBFT execution, $v = v'$.

- **Property 4:** Assume by contradiction that $sn.commit(r)$ occurred along with the other two linearization points of add. The linearization point of $sn.commit(r)$ occurs because of a quorum of VOTE(r, sn) messages sent by a set of processes P_1, and $sn.add(r', \_, r'')$ occurs because of a quorum of JOIN(r', sn) messages sent by a set of processes P_2. All the correct processes in P_1 must send VOTE(r, sn) messages to the leader of the next round therefore they must send VOTE(r_1, sn) to the leader of round r' where $r \leq r_1 < r'$. But $sn.add(r', \_, r'')$ shows that during round r', quorum of JOIN(r', sn) messages could be sent because all the correct processes in P_2 accepts that VOTE(r'', sn) is the message with the highest round (in ROUND−CHANGE(r') messages). Since P_1 and P_2 must have a an intersecting correct process and $r'' < r_1$, it is a contradiction.

□

3.8 Multi-Paxos (and its variants) Refines QTree

3.8.1 Description of the Protocol

Multi-Paxos runs a single-decree protocol instance like Paxos concurrently for each sequence number. Since Single-decree Paxos decides on one value, the easy way to agree on sequence of numbers would be to run Paxos multiple times for each sequence number. Multi-Paxos is more efficient version of such an approach. The main optimization in Multi-Paxos is to skip the first phase in Single-decree Paxos and not start a new round when the leader is stable. In other words, after the proposer updates $decidedVal$ with some decided value for sequence number sn in round r, it will directly propose some value for $sn+1$ in r. When the leader crashes, the next leader starts the round as Single-Decree Paxos but as an optimization, it sends a single START message for each sequence number simultaneously. Then processes send JOIN responses alongside with the highest voted rounds and values for each sequence number. The leader will continue with proposals one by one by selecting highest votes if there exist.

In the first phase, the leader broadcasts a generic START message to all the processes to start the round, executing the **START** action below, and processes acknowledge with a JOIN message if some conditions are met, executing the **JOIN** action below:

• **START Action:**

The leader p of round $r > 0$ (the proposer) broadcasts a START(r) message to all processes, waiting their highest votes for each sequence number sn.

• **JOIN Action:**

When a process p' receives a START(r) message, if p' has not sent a JOIN or VOTE message (explained below) for a higher round in the past, it replies by sending a JOIN(r) message to the proposer. This message includes maximum round numbers ($maxVotedRound$) of all sequence numbers for which p' has sent a VOTE message in the past and the value ($maxVotedValue$) proposed in that round. For each sequence number that it has not voted yet, these fields are 0 and $\perp$.

If the leader receives JOIN messages from a quorum of processes, the second phase starts. The leader broadcasts PROPOSE message for the next sequence number, executing the **PROPOSE** action below. Processes may acknowledge with a VOTE message if some conditions are met, executing a **VOTE** action. If the leader receives VOTE messages from a quorum of processes, then the proposed value becomes decided (and sent to the client) by executing a **DECIDE** action. When the leader decides on some value for sequence number sn without any failure in round r, the leader continues with PROPOSE action for $sn + 1$ in r. Otherwise a new round is inititated:

• **PROPOSE Action:**

When the proposer p receives JOIN(r) messages from a quorum of $(f + 1)$ processes, it selects the one with the highest vote round number for the current sequence number and proposes its value by broadcasting a PROPOSE(r, sn) message (which includes that value). If there is no such highest round (all vote rounds are 0), then the proposer selects the proposed value randomly simulating the value coming from the client (whose modeling we omit for simplicity).

• **VOTE Action:**

When a process p' receives a PROPOSE(r, sn) message, if p' has not sent a JOIN or VOTE message for a higher round in the past for sn, it replies by sending a VOTE(r, sn) message to the proposer with round number r and the same sequence number sn.

• **DECIDE Action:**

When the proposer p receives VOTE(r, sn) messages from a quorum of processes, it updates a local variable called $decidedVal[sn]$ to be the value it has proposed in this round r for sn. This assignment means that the value is decided and sent to the client after deciding for all sequence number $sn' < sn$.

Now we look at the descriptions of protocols which are variants of Multi-Paxos. The protocols that we consider in this section are Cheap Paxos [47], Stoppable Paxos [72], Fast Paxos [65] and Flexible Paxos [53].

Cheap Paxos is a variation of Multi-Paxos where additionally, f of the processes are idle as long as remaining $f + 1$ of them are the processes that generated the quorum in the first phase and remain alive. This optimization relies on the fact that, any leader can decide on sequence of values as long as all the processes in a fixed quorum are active. When there is a failure in this fixed quorum, current round ends and the new round starts after the crashed process is replaced with one of the idle process. Since there can be at most f faulty processes, there will be always (at least) one process which will exist in two consecutive quorums, not being idle.

Stoppable Paxos contains special stp command that can be proposed by a leader in some round for a sequence number sn and when this proposal is decided, no more commands are executed for sequence numbers $sn' > sn$. Since this variant enables to stop the current protocol and starts a new one using the final state, a replicated state machine can work as a sequence of stoppable state machines.

In Multi-Paxos, when there is no vote for some sequence number sn, the leader receives the value from the client and broadcasts to the processes. In Fast Paxos, when the first phase is skipped as in Multi-Paxos and the leader doesn't receive any vote for the current sequence number, the leader informs clients to send their request directly to all processes rather than to itself. The purpose of this approach is to reduce the end-to-end latency by allowing clients to send their requests directly to the processes but not through the leader (decreasing message delay). Then the processes send $fast$ votes according to the request that they receive and the leader decides on a value if there is a quorum of votes on the same value. In Fast Paxos, a quorum requires $2f + 1$ processes where the number of all processes is $3f + 1$. When the new round starts, for each sequence number, the leader select highest votes as listed below:

- If there is not a single vote, the leader selects the value to propose randomly.
- If there is only a single highest vote, the leader selects the value of that vote.
- If there are multiple votes, the one which is voted by $f + 1$ processes must be selected. If there is no such vote even though there are multiple votes, the leader selects the value of the proposal randomly.

Flexible Paxos is a variation of Multi-Paxos that allows different quorum sizes for first and second phases of the protocol as long as these two quorums intersect. Since the first phase is not executed as long as the leader is stable but the second phase is executed constantly, decreasing the number of processes to reach to a second quorum (also increasing the number of processes to reach to a first quorum), can increase the throughput by being capable of handling more failures.

3.8.2 Linearization Points in Multi-Paxos (and its variants)

We instrument Multi-Paxos with linearization points of successful QTree invocations. Multi-Paxos is a refinement of a set of QTree instances, one instance for each sequence number. The linearization points will refer to a specific instance identified using a sequence number, e.g., $sn.add(r, v, r')$ denotes an $add(r, v, r')$ invocation on the QTree instance sn. Therefore

- the linearization point of $sn.add(r, v, r')$ occurs when the proposer broadcasts the PROPOSE(r, sn) message containing value v (during the **PROPOSE** action in round r). v is the value of the JOIN(r, sn) message selected by the proposer. If $r' = 0$ then, v is selected randomly.
- the linearization point of $sn.commit(r)$ occurs when the proposer who is the leader of the round r updates $decidedVal$ for the sequence number sn.

Our main correctness theorem for Multi-Paxos (and its variants) is as follows:

Theorem 13. *Multi-Paxos (and its variants) refines a composition of independent QTree instances.*

Proof. We show that the sequence of successful add and $commit$ invocations on a QTree instance sn defined by linearization points along a Multi-Paxos execution satisfies the properties in Theorem 8 and therefore, it represents a correct QTree execution:

- **Property 1:** By definition, proposers can not propose two different proposals in the same round for the same sequence number. Since a leader can not propose for the next sequence number before deciding on the current one and round numbers are monotonically increasing when the leader is changed, the linearization point of $sn.add(r, \_, \_)$ will occur at most once for a round r with the same sequence number sn. Therefore, processes can only vote for a single propose with the same round and sequence number. This implies that at most one linearization point of $sn.commit(r)$ can occur for a round r with the same sequence number sn.
- **Property 2:** This holds trivially as all the processes follow the rules of the protocol and they need to receive a PROPOSE(r, sn) message (which can occur only after the linearization point of $sn.add(r, \_, \_)$) from the leader of the current round to send VOTE(r, sn) message.
- **Property 3:** In Multi-Paxos, leaders select the value that will be proposed for each sequence number separately, it can be accepted as running different instances **PROPOSE** action of Single-decree Paxos. Therefore, the proof will follow as the proof of Property 3 in Section 3.4.1.2, by considering that the property holds for each sequence number sn. Note

that, skipping first phases in a round for the next sequence numbers after the first decision under a stable leader does not affect the proof because at the beginning of this round, the leader has already received highest votes (if there exist) from a quorum of processes for all sequence numbers.

  - **Property 3a:** It holds by the proof of Property 3a in Section 3.4.1.2, by considering it for each sn.

- **Property 4:** Assume by contradiction that $sn.commit(r)$ occurred along with the other two linearization points of add. The linearization point of $sn.commit(r)$ occurs because of a quorum of VOTE(r, sn) messages sent by a set of processes P_1, and $sn.add(r', \_, r'')$ because of a quorum of JOIN(r') messages sent by a set of processes P_2. Since P_1 and P_2 must have a non-empty intersection, by the definition of the **JOIN** action, it must be the case that $r'' \geq r$, which contradicts the hypothesis.

□

Cheap Paxos and Stoppable Paxos are just restricted versions of Multi-Paxos in which indices of sequences are independent. Therefore Multi-Paxos and both protocols refine QTree. Since the quorums are independent from which processes are included, Multi-Paxos will work the same with Cheap Paxos using the same processes. As a side note, in Cheap Paxos, since quorum of active (not idle) processes and any quorum decided a value must intersect, these values will be propagated the next rounds without any problem. In Stoppable Paxos, a decided value for a sequence number cannot turn into undecided or change its value due to a decided stp value in another sequence number. Decided stp value can only prevent execution on a higher sequence number. Since both Multi-Paxos and Stoppable Paxos progress the same until a stp command is executed, Multi-Paxos and therefore Stoppable Paxos refine a set of QTree instances.

In Fast Paxos, when the processes receive request directly from the clients and propose accordingly, there can be multiple proposals due to network which are not proposed by the leader. Therefore we need to redefine the linearization points for Fast Paxos:

- the linearization point of $sn.add(r, v, r')$ occurs when the leader broadcasts the PROPOSE(r, sn) message containing value v (during the **PROPOSE** action in round r). v is the value of the JOIN(r, sn) message selected by the proposer. If $r' = 0$ then v is selected randomly. If the proposer is not the leader, then $sn.add(r, v, r')$ occurs when the leader of the round r updates $decidedVal$ with value v fo sequence number sn, after obtaining a quorum of votes for v. Note that $r' = 0$ and v is selected randomly as no highest vote seen by the leader, for the sequence number sn.

- the linearization point of $sn.commit(r)$ occurs when the proposer who is the leader of the round r updates $decidedVal$ with value v fo sequence number sn.

Simply, when the processes receive the proposal from the client directly, the linearization point of $sn.add(r, v, r')$ occurs at the same time with $sn.commit(r)$ (if it is added) according to the definition of $sn.commit(r)$. This is also intuitive and shows how this protocol received his name. Properties still hold because it is ensured that there was no vote beforehand for current sequence number if the request is directly received from the client and votes from these rounds are not considered when a new round starts:

- **Property 1:** Property 1 in Multi-Paxos holds. Additionally, when the proposal is directly received from the client, as there can be only 1 quorum of fast votes for this proposal in the same round, there can be at most one linearization point of $sn.add(r, \_, \_)$ and at most one linearization point of $sn.commit(r)$.
- **Property 2:** It holds by the proof of Property 2 in Multi-Paxos.
- **Property 3:** It holds by the proof of Property 3 in Multi-Paxos. When the linearization point of $sn.add(r, \_, \_)$ occurs after the proposal that is received by a process directly from the client, since $r' = 0$, this property is not considered.
  - **Property 3a:** It holds by the proof of Property 3a in Multi-Paxos. Again, when the linearization point of $sn.add(r, \_, \_)$ occurs after the proposal that is received by a process directly from the client, since $r' = 0$, this property is not considered.
- **Property 4:** Assume by contradiction that $sn.commit(r)$ occurred along with the other two linearization points of add. The linearization point of $sn.commit(r)$ occurs because of a quorum of $\texttt{VOTE}(r)$ messages sent by a set of processes P_1, and $sn.add(r', \_, r'')$ exists because:
  - There was only one value in highest votes for sn (which is voted in round r') in $\texttt{JOIN}(r')$ messages sent by a set of processes P_2 or
  - There were multiple values in highest votes (which are voted in round r') but one of them is voted by at least $f + 1$ of the processes in P_2.

  In both cases, since P_1 and P_2 must have $f + 1$ processes intersecting, by the definition of the **JOIN** action, it must be the case that $r'' \geq r$, which contradicts the hypothesis.

Flexible Paxos refines QTree as the proof of properties perfectly fit for this protocol, without a modification on linearization points. In proof of properties for Multi-Paxos, the only two quorum that we are interested on their intersection is quorums from first and second phases (quorum

of JOIN and VOTE respectively, used in Property 4). Therefore, changing their sizes as long as they intersect doesn't affect the proof and the proof for Multi-Paxos holds as it is for Flexible Paxos.

3.9 Related Work

The problem of proving the correctness of such protocols has been studied in previous work. We give an overview of the existing approaches that starts with safety proof methods based on refinement, which are closer to our approach.

Refinement based safety proofs. Verdi [100] is a framework for implementing and verifying distributed systems that contains formalizations of various network semantics and failure models. Verdi provides *system transformers* useful for refining high-level specifications to concrete implementations. As a case study, it includes a fully-mechanized correctness proof of Raft [102]. This proof consists of 45000 lines of proof code (manual annotations) in the Coq language for a 5000 lines RAFT implementation, showing the difficulty of reasoning on consensus protocols and the manual effort required. Iron Fleet [49] uses TLA [66] style transition-system specifications and refine them to low-level implementations described in the Dafny programming language [69]. Boichat et al. [9] defines a class of specifications for consensus protocols, which are more abstract than QTree and can make correctness proofs harder. Proving Paxos in their case is reduced to a linearizability proof towards an abstract specification, which is quite complex because the linearization points are *not fixed*. As a possibly superficial quantitative measure, their Paxos proof reduces to 7 lemmas that are formalized by Garcia-Perez et al. [36, 37] in 12 pages (see Appendix B and C in [37]), much more than our QTree proof. Our refinement proof is also similar to a linearizability proof, but the linearization points in our case are fixed (do not depend on the future of an execution) which brings more simplicity. In principle, the specifications in [9] could apply to more protocols, but we are not aware of such a case. The inductive sequentialization proof rule [63] is used for a fully mechanized correctness proof of a realistic Paxos implementation. This implementation is proved to be a refinement of a *sequential* program whose definition is quite close to the original implementation, much less abstract than QTree, and relies on commutativity arguments implied by the communication-closed round structure [27]. A similar idea is explored in [38], but in a more restricted context.

Inductive invariant based safety proofs. Ivy [79] is an SMT-based safety verification tool that can be used for verifying inductive invariants about global states of a distributed protocol. In order to stay in a decidable fragment of first-order logic, both the modeling and the specification language of IVY are restricted. A simple model of Paxos obeying these restrictions is proven correct in [78].

Beyond safety. The TLA+ infrastructure [66] of Lamport has been used to verify both safety and liveness (termination) of several variations of Paxos, e.g., Fast Paxos [65] or Multi-Paxos [18]. Bravo et al. [11] introduce a generic synchronization mechanism for round changes, called the view synchronizer, which guarantees liveness for various Byzantine consensus protocols including our cases studies HotStuff and PBFT. This work includes full correctness proofs for single-decree versions of HotStuff and PBFT and a two-phase version of HotStuff. PSync [26] provides a partially synchronous semantics for distributed protocols assuming communication-closed rounds in the Heard-Of model [20]. PSync is used to prove both safety and liveness of a Paxos-like consensus protocol called *lastVoting*.

Relating different consensus protocols. Lamport defines a series of refinements of Paxos that leads to a Byzantine fault tolerant version, which is refined by PBFT [64]. Our proof that Paxos refines QTree can be easily extended to this Byzantine fault tolerant version in the same manner as we did for PBFT. Wang et al. [98] shows that a variation of RAFT is a refinement of Paxos, which enables porting some Paxos optimizations to RAFT. Renesse et al. [86] compare Paxos, Viewstamped Replication [77] and ZAB [58]. They define a rooted tree of specifications represented in TLA style whose leaves are concrete protocols. Each node in this tree is refined by its children. Common ancestors of concrete protocols show similarities whereas conflicting specifications show the differences. Similarly, [91] shows that Paxos, Chandra-Toueg [19] and Ben-Or [7] consensus algorithms share common building blocks. Aublin et al. [6] propose an abstract data type for specifying existing and possible future consensus protocols. Unlike our QTree, core components of this data type are not implemented and intentionally left abstract so that it can adapt to different network and process failure models.

4 Conclusion

In this dissertation, we explored different approaches to improve reliability of concurrent data structures built either on top of shared memory or message passing. We provided several algorithms for finding and fixing linearizability violations in shared-memory concurrent data structures. For discovering these violations more efficiently, we proposed several stateful model checking algorithms based on POR that focus on overall time performance. Next, we offered a novel approach for root causing the lineraziability violations by suggesting repairs in the form of atomic code segments in the source code of the library that allow for maximum concurrency. Then we turned our direction to distributed systems, specifically to consensus protocols, and we proposed a uniform abstraction called QTree for showing that various types of protocols are safe. Looking at these contributions in more detail:

- Section 2.2 presents three stateful POR algorithms: one basic static algorithm named S-POR, and two novel algorithms that compute the recently proposed source sets dynamically, called DE-S-POR and DL-S-POR which are built on top of S-POR. Our algorithms focus on overall performance instead of theoretical optimality. To evaluate, we compared them with (1) their variations that are built on top of the standard setup of JPF, (2) O-DPOR (stateful version of Source-DPOR in [3]) and (3) their stateless variations. We showed that even though DL-S-POR is not an optimal algorithm, it outperforms all the other algorithms when there is high potential for reduction and otherwise, S-POR is faster due to its simple nature which suggest running both algorithms in parallel in the context of a portfolio model checker. Then we also examined how the enumeration of transitions affects the performance of DL-S-POR and S-POR for finding bugs and we concluded that selecting the next transition uniformly at random is better in average but if there is enough parallelism, then using a portfolio model checker where there will be parallel runs for each permutation of thread ids would outperform another portfolio model checker based on randomization with different seeds.

- Section 2.3 presents a novel technique for root-causing linearizability violations by identifying minimal code blocks, called optimal repairs, that must be executed without any interference from other threads in order to eliminate those violations. We showed that the

algorithm for identifying optimal repairs works in polynomial time when the number of threads is fixed. We define an heuristic to select the best optimal repairs from many candidates. To check the efficacy of our approach, we generated non-linearizable variations for several lock-based libraries by shrinking their atomic sections and showed that the obtained results as optimal repairs are precise as they mostly cover the positions of original locks.

- Chapter 3 describes a new methodology for proving safety of consensus or state-machine replication protocols, which relies on a novel abstraction of their dynamics. This abstraction is defined as a sequential QTree object whose state represents a global view of a protocol execution and its operations correspond to some process doing a step that witnesses for the receipt a quorum of messages. These operations of QTree construct a tree structure and model agreement on values or a sequence of state-machine commands as agreement on a fixed branch in the tree. Our methodology applies uniformly to a range of protocols and the ones considered in this chapter can be grouped in three classes: single-decree consensus (Paxos), multi-decree consensus (PBFT, Multi-Paxos) and state machine replication (Raft, HotStuff)[1]. We showed that they all refine QTree: a single instance for Paxos and HotStuff, and a set of independent instances (one for each sequence number in a command log) for PBFT, Multi-Paxos, and Raft. We believe that this QTree helps in improving the understanding of such protocols and writing correct implementations or optimizations thereof.

4.1 Future Work

The results in this thesis suggest several possible directions for future work:

- Chapter 2 assumes that programs run under sequential consistency. Modern processors or programming languages work under weak memory models such as TSO, and extending these algorithms to cover such semantics is a possible avenue for further research.

- In Section 2.2, we compared stateful POR algorithms with stateless ones and stateful versions of the algorithms outperform their stateless variations in terms of time. This happens in the context of a tradeoff between space and time, since stateful versions consume much less memory. One possible direction for future work is investigating ”hybrid” algorithms that are in between stateless and stateful model checking to find a sweet-spot where the overall gain is maximized.

- Moreover, our POR algorithms are defined in the context of explicit-state model checking. Evaluating their variations in the context of symbolic model checking where sets of states

[1] This is a slight abuse of terminology since multi-decree consensus protocols are typically used to implement state machine replication.

are maintained symbolically and not in an explicit manner is an interesting direction to investigate.

- In Section 2.3, we are describing an independent component for searching execution eliminators where the inputs are non-linearizable executions of some client of the data structure. This relies on running the model checker first to identify the non-linearizable executions and then, apply this component. We would like to investigate whether merging the two can improve overall performance.

- In Chapter 3, we presented QTree for proving that consensus protocols such as Paxos are safe. As future work, we might explore the use of QTree in reasoning about liveness. This would require some fairness condition on infinite sequences of add/commit invocations, and a suitable notion of refinement which ensures that infinite sequences of protocol steps cannot be mapped to infinite sequences of stuttering QTree steps.

- Most of the consensus protocols (also the ones considered in this dissertation) behave in a sequence of ordered rounds. There are however protocols such as Texel [85] which do not admit such a decomposition in rounds. An interesting direction for future work is to investigate whether QTree applies or can be extended to protocols without the concept of rounds. Another such assumption of these protocols is that the rounds are communication-closed and again checking protocols that do not assume it can be an interesting direction to explore.

Bibliography

1. P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. "Stateless model checking for TSO and PSO". *Acta Informatica* 54:8, 2017, pp. 789–818. DOI: 10.1007/s00236-016-0275-0. URL: https://doi.org/10.1007/s00236-016-0275-0.

2. P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. "Comparing Source Sets and Persistent Sets for Partial Order Reduction". In: *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*. Ed. by L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, and R. Mardare. Vol. 10460. Lecture Notes in Computer Science. Springer, 2017, pp. 516–536. DOI: 10.1007/978-3-319-63121-9_26. URL: https://doi.org/10.1007/978-3-319-63121-9%5C_26.

3. P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. "Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction". *J. ACM* 64:4, 2017, 25:1–25:49. DOI: 10.1145/3073408. URL: https://doi.org/10.1145/3073408.

4. P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo. "Optimal stateless model checking under the release-acquire semantics". *Proc. ACM Program. Lang.* 2:OOPSLA, 2018, 135:1–135:29. DOI: 10.1145/3276505. URL: https://doi.org/10.1145/3276505.

5. Y. Afek, E. Gafni, and A. Morrison. "Common2 extended to stacks and unbounded concurrency". *Distributed Computing* 20:4, 2007, pp. 239–252. DOI: 10.1007/s00446-007-0023-3. URL: https://doi.org/10.1007/s00446-007-0023-3.

6. P. Aublin, R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. "The Next 700 BFT Protocols". *ACM Trans. Comput. Syst.* 32:4, 2015, 12:1–12:45. DOI: 10.1145/2658994. URL: https://doi.org/10.1145/2658994.

7. M. Ben-Or. "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract)". In: *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*. Ed. by R. L. Probert, N. A. Lynch, and N. Santoro. ACM, 1983, pp. 27–30.

DOI: 10.1145/800221.806707. URL: https://doi.org/10.1145/800221.806707.

8. R. Bloem, G. Hofferek, B. Könighofer, R. Könighofer, S. Ausserlechner, and R. Spork. "Synthesis of synchronization using uninterpreted functions". In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 2014, pp. 35–42. DOI: 10.1109/FMCAD.2014.6987593. URL: https://doi.org/10.1109/FMCAD.2014.6987593.

9. R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. "Deconstructing paxos". *SIGACT News* 34:1, 2003, pp. 47–67. DOI: 10.1145/637437.637447. URL: https://doi.org/10.1145/637437.637447.

10. A. Bouajjani and M. Emmi. "Bounded Phase Analysis of Message-Passing Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by C. Flanagan and B. König. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 451–465. DOI: 10.1007/978-3-642-28756-5_31. URL: https://doi.org/10.1007/978-3-642-28756-5%5C_31.

11. M. Bravo, G. V. Chockler, and A. Gotsman. "Making Byzantine Consensus Live". In: *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*. Ed. by H. Attiya. Vol. 179. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 23:1–23:17. DOI: 10.4230/LIPIcs.DISC.2020.23. URL: https://doi.org/10.4230/LIPIcs.DISC.2020.23.

12. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. "Line-up: a complete and automatic linearizability checker". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by B. G. Zorn and A. Aiken. ACM, 2010, pp. 330–340. DOI: 10.1145/1806596.1806634. URL: https://doi.org/10.1145/1806596.1806634.

13. J. Burnim, G. C. Necula, and K. Sen. "Specifying and checking semantic atomicity for multithreaded programs". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. Ed. by R. Gupta and T. C. Mowry. ACM, 2011, pp. 79–90. DOI: 10.1145/1950365.1950377. URL: https://doi.org/10.1145/1950365.1950377.

14. M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*. Ed. by M. I. Seltzer and P. J. Leach. USENIX Association, 1999, pp. 173–186. URL: https://dl.acm.org/citation.cfm?id=296824.

15. P. Cerný, E. M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach. "From Non-preemptive to Preemptive Scheduling Using Synchronization Synthesis". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Ed. by D. Kroening and C. S. Pasareanu. Vol. 9207. Lecture Notes in Computer Science. Springer, 2015, pp. 180–197. DOI: 10.1007/978-3-319-21668-3_11. URL: https://doi.org/10.1007/978-3-319-21668-3%5C_11.

16. P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. "Efficient Synthesis for Concurrency by Semantics-Preserving Transformations". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by N. Sharygina and H. Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 951–967. DOI: 10.1007/978-3-642-39799-8_68. URL: https://doi.org/10.1007/978-3-642-39799-8%5C_68.

17. P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. "Regression-Free Synthesis for Concurrency". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by A. Biere and R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 568–584. DOI: 10.1007/978-3-319-08867-9_38. URL: https://doi.org/10.1007/978-3-319-08867-9%5C_38.

18. S. Chand, Y. A. Liu, and S. D. Stoller. "Formal Verification of Multi-Paxos for Distributed Consensus". In: *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. Ed. by J. S. Fitzgerald, C. L. Heitmeyer, S. Gnesi, and A. Philippou. Vol. 9995. Lecture Notes in Computer Science. 2016, pp. 119–136. DOI: 10.1007/978-3-319-48989-6_8. URL: https://doi.org/10.1007/978-3-319-48989-6%5C_8.

19. T. D. Chandra and S. Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems". *J. ACM* 43:2, 1996, pp. 225–267. DOI: 10.1145/226643.226647. URL: https://doi.org/10.1145/226643.226647.

20. B. Charron-Bost and S. Merz. "Formal Verification of a Consensus Algorithm in the Heard-Of Model". *Int. J. Softw. Informatics* 3:2-3, 2009, pp. 273–303. URL: http://www.ijsi.org/ch/reader/view%5C_abstract.aspx?file%5C_no=273%5C&flag=1.

21. S. Cherem, T. M. Chilimbi, and S. Gulwani. "Inferring locks for atomic sections". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by R. Gupta and S. P. Amarasinghe. ACM, 2008, pp. 304–315. DOI: 10.1145/1375581.1375619. URL: https://doi.org/10.1145/1375581.1375619.

22. L. Chew and D. Lie. "Kivati: fast detection and prevention of atomicity violations". In: *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*. Ed. by C. Morin and G. Muller. ACM, 2010, pp. 307–320. DOI: 10.1145/1755913.1755945. URL: https://doi.org/10.1145/1755913.1755945.

23. E. M. Clarke and E. A. Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic". In: *25 Years of Model Checking - History, Achievements, Perspectives*. Ed. by O. Grumberg and H. Veith. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 196–215. DOI: 10.1007/978-3-540-69850-0_12. URL: https://doi.org/10.1007/978-3-540-69850-0%5C_12.

24. E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach". In: *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*. Ed. by J. R. Wright, L. Landweber, A. J. Demers, and T. Teitelbaum. ACM Press, 1983, pp. 117–126. DOI: 10.1145/567067.567080. URL: https://doi.org/10.1145/567067.567080.

25. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking, 1st Edition*. MIT Press, 2001. ISBN: 978-0-262-03270-4. URL: http://books.google.de/books?id=Nmc4wEaLXFEC.

26. C. Dragoi, T. A. Henzinger, and D. Zufferey. "PSync: a partially synchronous language for fault-tolerant distributed algorithms". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by R. Bodik and R. Majumdar. ACM, 2016, pp. 400–415. DOI: 10.1145/2837614.2837650. URL: https://doi.org/10.1145/2837614.2837650.

27. T. Elrad and N. Francez. "Decomposition of Distributed Programs into Communication-Closed Layers". *Sci. Comput. Program.* 2:3, 1982, pp. 155–173. DOI: 10.1016/0167-6423(83)90013-8. URL: https://doi.org/10.1016/0167-6423(83)90013-8.

28. M. Emmi and C. Enea. "Exposing Non-Atomic Methods of Concurrent Objects". *CoRR* abs/1706.09305, 2017. arXiv: 1706.09305. URL: http://arxiv.org/abs/1706.09305.

29. M. Emmi and C. Enea. "Violat: Generating Tests of Observational Refinement for Concurrent Objects". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Ed. by I. Dillig and S. Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 534–546. DOI: 10.1007/978-3-030-25543-5_30. URL: https://doi.org/10.1007/978-3-030-25543-5%5C_30.

30. M. Emmi and C. Enea. "Weak-consistency specification via visibility relaxation". *PACMPL* 3:POPL, 2019, 60:1–60:28. DOI: 10.1145/3290373. URL: https://doi.org/10.1145/3290373.

31. M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. "Lock allocation". In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. Ed. by M. Hofmann and M. Felleisen. ACM, 2007, pp. 291–296. DOI: 10.1145/1190216.1190260. URL: https://doi.org/10.1145/1190216.1190260.

32. M. Emmi, S. Qadeer, and Z. Rakamaric. "Delay-bounded scheduling". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by T. Ball and M. Sagiv. ACM, 2011, pp. 411–422. DOI: 10.1145/1926385.1926432. URL: https://doi.org/10.1145/1926385.1926432.

33. D. R. Engler and K. Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. Ed. by M. L. Scott and L. L. Peterson. ACM, 2003, pp. 237–252. DOI: 10.1145/945445.945468. URL: https://doi.org/10.1145/945445.945468.

34. A. Farzan and P. Madhusudan. "Monitoring Atomicity in Concurrent Programs". In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. Ed. by A. Gupta and S. Malik. Vol. 5123. Lecture Notes

in Computer Science. Springer, 2008, pp. 52–65. DOI: 10.1007/978-3-540-70545-1_8. URL: https://doi.org/10.1007/978-3-540-70545-1%5C_8.

35. C. Flanagan and P. Godefroid. "Dynamic partial-order reduction for model checking software". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. Ed. by J. Palsberg and M. Abadi. ACM, 2005, pp. 110–121. DOI: 10.1145/1040305.1040315. URL: https://doi.org/10.1145/1040305.1040315.

36. A. Garcia-Perez, A. Gotsman, Y. Meshman, and I. Sergey. "Paxos Consensus, Deconstructed and Abstracted". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by A. Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 912–939. DOI: 10.1007/978-3-319-89884-1_32. URL: https://doi.org/10.1007/978-3-319-89884-1%5C_32.

37. A. Garcia-Perez, A. Gotsman, Y. Meshman, and I. Sergey. "Paxos Consensus, Deconstructed and Abstracted (Extended Version)". *CoRR* abs/1802.05969, 2018. arXiv: 1802.05969. URL: http://arxiv.org/abs/1802.05969.

38. K. von Gleissenthall, R. G. Kici, A. Bakst, D. Stefan, and R. Jhala. "Pretend synchrony: synchronous verification of asynchronous distributed programs". *Proc. ACM Program. Lang.* 3:POPL, 2019, 59:1–59:30. DOI: 10.1145/3290372. URL: https://doi.org/10.1145/3290372.

39. P. Godefroid. "Model Checking for Programming Languages using Verisoft". In: *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*. Ed. by P. Lee, F. Henglein, and N. D. Jones. ACM Press, 1997, pp. 174–186. DOI: 10.1145/263699.263717. URL: https://doi.org/10.1145/263699.263717.

40. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Vol. 1032. 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.

41. P. Godefroid. "Using Partial Orders to Improve Automatic Verification Methods". In: *Computer-Aided Verification, Proceedings of a DIMACS Workshop 1990, New Brunswick, New Jersey, USA, June 18-21, 1990*. Ed. by E. M. Clarke and R. P. Kurshan. Vol. 3. DIMACS

Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1990, pp. 321–340. DOI: 10.1090/dimacs/003/21. URL: https://doi.org/10.1090/dimacs/003/21.

42. P. Godefroid, G. J. Holzmann, and D. Pirottin. "State-Space Caching Revisited". *Formal Methods Syst. Des.* 7:3, 1995, pp. 227–241. DOI: 10.1007/BF01384077. URL: https://doi.org/10.1007/BF01384077.

43. P. Godefroid and D. Pirottin. "Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract)". In: *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. Ed. by C. Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 438–449. DOI: 10.1007/3-540-56922-7_36. URL: https://doi.org/10.1007/3-540-56922-7%5C_36.

44. P. Godefroid and P. Wolper. "Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties". *Formal Methods Syst. Des.* 2:2, 1993, pp. 149–164. DOI: 10.1007/BF01383879. URL: https://doi.org/10.1007/BF01383879.

45. W. M. Golab, L. Higham, and P. Woelfel. "Linearizable implementations do not suffice for randomized distributed computation". In: *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*. Ed. by L. Fortnow and S. P. Vadhan. ACM, 2011, pp. 373–382. DOI: 10.1145/1993636.1993687. URL: https://doi.org/10.1145/1993636.1993687.

46. V. Gramoli. "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*. Ed. by A. Cohen and D. Grove. ACM, 2015, pp. 1–10. DOI: 10.1145/2688500.2688501. URL: https://doi.org/10.1145/2688500.2688501.

47. J. Gray and L. Lamport. "Consensus on Transaction Commit". *CoRR* cs.DC/0408036, 2004. URL: http://arxiv.org/abs/cs.DC/0408036.

48. A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach. "Succinct Representation of Concurrent Trace Sets". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by S. K. Rajamani and D. Walker. ACM, 2015, pp. 433–444. DOI: 10.1145/2676726.2677008. URL: https://doi.org/10.1145/2676726.2677008.

49. C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. "IronFleet: proving practical distributed systems correct". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by E. L. Miller and S. Hand. ACM, 2015, pp. 1–17. DOI: 10.1145/2815400.2815428. URL: https://doi.org/10.1145/2815400.2815428.

50. M. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". *ACM Trans. Program. Lang. Syst.* 12:3, 1990, pp. 463–492. DOI: 10.1145/78969.78972. URL: https://doi.org/10.1145/78969.78972.

51. G. J. Holzmann and D. A. Peled. "An improvement in formal verification". In: *Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994*. Ed. by D. Hogrefe and S. Leue. Vol. 6. IFIP Conference Proceedings. Chapman & Hall, 1994, pp. 197–211.

52. A. Horn and D. Kroening. "Faster Linearizability Checking via P-Compositionality". In: *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*. Ed. by S. Graf and M. Viswanathan. Vol. 9039. Lecture Notes in Computer Science. Springer, 2015, pp. 50–65. DOI: 10.1007/978-3-319-19195-9_4. URL: https://doi.org/10.1007/978-3-319-19195-9%5C_4.

53. H. Howard, D. Malkhi, and A. Spiegelman. "Flexible Paxos: Quorum intersection revisited". *CoRR* abs/1608.06696, 2016. arXiv: 1608.06696. URL: http://arxiv.org/abs/1608.06696.

54. J. Huang and C. Zhang. "An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs". In: *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. Ed. by E. Yahav. Vol. 6887. Lecture Notes in Computer Science. Springer, 2011, pp. 163–179. DOI: 10.1007/978-3-642-23702-7_15. URL: https://doi.org/10.1007/978-3-642-23702-7%5C_15.

55. N. Jalbert and K. Sen. "A trace simplification technique for effective debugging of concurrent programs". In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. Ed. by G. Roman and A. van der Hoek. ACM, 2010, pp. 57–66. DOI: 10.1145/1882291.1882302. URL: https://doi.org/10.1145/1882291.1882302.

56. G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. "Automated atomicity-violation fixing". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by M. W. Hall and D. A. Padua. ACM, 2011, pp. 389–400. DOI: 10.1145/1993498.1993544. URL: https://doi.org/10.1145/1993498.1993544.

57. G. Jin, W. Zhang, and D. Deng. "Automated Concurrency-Bug Fixing". In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. Ed. by C. Thekkath and A. Vahdat. USENIX Association, 2012, pp. 221–236. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/jin.

58. F. P. Junqueira, B. C. Reed, and M. Serafini. "Zab: High-performance broadcast for primary-backup systems". In: *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*. IEEE Compute Society, 2011, pp. 245–256. DOI: 10.1109/DSN.2011.5958223. URL: https://doi.org/10.1109/DSN.2011.5958223.

59. S. Kashyap and V. K. Garg. "Producing Short Counterexamples Using "Crucial Events"". In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. Ed. by A. Gupta and S. Malik. Vol. 5123. Lecture Notes in Computer Science. Springer, 2008, pp. 491–503. DOI: 10.1007/978-3-540-70545-1_47. URL: https://doi.org/10.1007/978-3-540-70545-1%5C_47.

60. S. Katz and D. A. Peled. "Verification of Distributed Programs Using Representative Interleaving Sequences". *Distributed Comput.* 6:2, 1992, pp. 107–120. DOI: 10.1007/BF02252682. URL: https://doi.org/10.1007/BF02252682.

61. M. Kokologiannakis and V. Vafeiadis. "GenMC: A Model Checker for Weak Memory Models". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by A. Silva and K. R. M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 427–440. DOI: 10.1007/978-3-030-81685-8_20. URL: https://doi.org/10.1007/978-3-030-81685-8%5C_20.

62. M. Kokologiannakis and V. Vafeiadis. "HMC: Model Checking for Hardware Memory Models". In: *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. Ed. by J. R. Larus, L. Ceze, and K. Strauss. ACM, 2020, pp. 1157–1171. DOI: 10.1145/3373376.3378480. URL: https://doi.org/10.1145/3373376.3378480.

63. B. Kragl, C. Enea, T. A. Henzinger, S. O. Mutluergil, and S. Qadeer. "Inductive sequentialization of asynchronous programs". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by A. F. Donaldson and E. Torlak. ACM, 2020, pp. 227–242. DOI: 10.1145/3385412.3385980. URL: https://doi.org/10.1145/3385412.3385980.

64. L. Lamport. "Byzantizing Paxos by Refinement". In: *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*. Ed. by D. Peleg. Vol. 6950. Lecture Notes in Computer Science. Springer, 2011, pp. 211–224. DOI: 10.1007/978-3-642-24100-0_22. URL: https://doi.org/10.1007/978-3-642-24100-0%5C_22.

65. L. Lamport. "Fast Paxos". *Distributed Comput.* 19:2, 2006, pp. 79–103. DOI: 10.1007/s00446-006-0005-x. URL: https://doi.org/10.1007/s00446-006-0005-x.

66. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X. URL: http://research.microsoft.com/users/lamport/tla/book.html.

67. L. Lamport. "The Part-Time Parliament". *ACM Trans. Comput. Syst.* 16:2, 1998, pp. 133–169. DOI: 10.1145/279227.279229. URL: https://doi.org/10.1145/279227.279229.

68. S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. "Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques". In: *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by D. S. Rosenblum and G. Taentzer. Vol. 6013. Lecture Notes in Computer Science. Springer, 2010, pp. 308–322. DOI: 10.1007/978-3-642-12029-9_22. URL: https://doi.org/10.1007/978-3-642-12029-9%5C_22.

69. K. R. M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Ed. by E. M. Clarke and A. Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20. URL: https://doi.org/10.1007/978-3-642-17511-4%5C_20.

70. P. Liu, O. Tripp, and X. Zhang. "Flint: fixing linearizability violations". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. Ed. by A. P. Black and T. D. Millstein. ACM, 2014, pp. 543–560. DOI: 10.1145/2660193.2660217. URL: https://doi.org/10.1145/2660193.2660217.

71. G. Lowe. "Testing for linearizability". *Concurrency and Computation: Practice and Experience* 29:4, 2017. DOI: 10.1002/cpe.3928. URL: https://doi.org/10.1002/cpe.3928.

72. D. Malkhi, L. Lamport, and L. Zhou. *Stoppable Paxos*. Technical report MSR-TR-2008-192. 2008.

73. Z. Manna and P. Wolper. "Synthesis of Communicating Processes from Temporal Logic Specifications". *ACM Trans. Program. Lang. Syst.* 6:1, 1984, pp. 68–93. DOI: 10.1145/357233.357237. URL: https://doi.org/10.1145/357233.357237.

74. A. W. Mazurkiewicz. "Trace Theory". In: *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*. Ed. by W. Brauer, W. Reisig, and G. Rozenberg. Vol. 255. Lecture Notes in Computer Science. Springer, 1986, pp. 279–324. DOI: 10.1007/3-540-17906-2_30. URL: https://doi.org/10.1007/3-540-17906-2%5C_30.

75. M. Musuvathi and S. Qadeer. "Iterative context bounding for systematic testing of multithreaded programs". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by J. Ferrante and K. S. McKinley. ACM, 2007, pp. 446–455. DOI: 10.1145/1250734.1250785. URL: https://doi.org/10.1145/1250734.1250785.

76. T. Neele, A. Wijs, D. Bosnacki, and J. van de Pol. "Partial-Order Reduction for GPU Model Checking". In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Ed. by C. Artho, A. Legay, and D. Peled. Vol. 9938. Lecture Notes in Computer Science. 2016, pp. 357–374. DOI: 10.1007/978-3-319-46520-3_23. URL: https://doi.org/10.1007/978-3-319-46520-3%5C_23.

77. B. M. Oki and B. Liskov. "Viewstamped Replication: A General Primary Copy". In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*. Ed. by D. Dolev. ACM, 1988, pp. 8–17.

DOI: 10.1145/62546.62549. URL: https://doi.org/10.1145/62546.62549.

78. O. Padon, G. Losa, M. Sagiv, and S. Shoham. "Paxos made EPR: decidable reasoning about distributed protocols". *Proc. ACM Program. Lang.* 1:OOPSLA, 2017, 108:1–108:31. DOI: 10.1145/3140568. URL: https://doi.org/10.1145/3140568.

79. O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. "Ivy: safety verification by interactive generalization". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by C. Krintz and E. D. Berger. ACM, 2016, pp. 614–630. DOI: 10.1145/2908080.2908118. URL: https://doi.org/10.1145/2908080.2908118.

80. C. H. Papadimitriou. "The serializability of concurrent database updates". *J. ACM* 26:4, 1979, pp. 631–653. DOI: 10.1145/322154.322158. URL: https://doi.org/10.1145/322154.322158.

81. S. Park, S. Lu, and Y. Zhou. "CTrigger: exposing atomicity violation bugs from their hiding places". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*. Ed. by M. L. Soffa and M. J. Irwin. ACM, 2009, pp. 25–36. DOI: 10.1145/1508244.1508249. URL: https://doi.org/10.1145/1508244.1508249.

82. D. A. Peled. "All from One, One for All: on Model Checking Using Representatives". In: *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. Ed. by C. Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 409–423. DOI: 10.1007/3-540-56922-7_34. URL: https://doi.org/10.1007/3-540-56922-7%5C_34.

83. S. Qadeer and J. Rehof. "Context-Bounded Model Checking of Concurrent Software". In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by N. Halbwachs and L. D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 93–107. DOI: 10.1007/978-3-540-31980-1_7. URL: https://doi.org/10.1007/978-3-540-31980-1%5C_7.

84. J. Queille and J. Sifakis. "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8,*

1982, Proceedings. Ed. by M. Dezani-Ciancaglini and U. Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351. DOI: 10.1007/3-540-11494-7_22. URL: https://doi.org/10.1007/3-540-11494-7%5C_22.

85. R. van Renesse. "Asynchronous Consensus Without Rounds". *CoRR* abs/1908.10716, 2019. arXiv: 1908.10716. URL: http://arxiv.org/abs/1908.10716.

86. R. van Renesse, N. Schiper, and F. B. Schneider. "Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab". *IEEE Trans. Dependable Secur. Comput.* 12:4, 2015, pp. 472–484. DOI: 10.1109/TDSC.2014.2355848. URL: https://doi.org/10.1109/TDSC.2014.2355848.

87. R. L. Rivest, A. Shamir, and L. M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (Reprint)". *Commun. ACM* 26:1, 1983, pp. 96–99. DOI: 10.1145/357980.358017. URL: https://doi.org/10.1145/357980.358017.

88. P. Rogaway and T. Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". *IACR Cryptol. ePrint Arch.*, 2004, p. 35. URL: http://eprint.iacr.org/2004/035.

89. M. Said, C. Wang, Z. Yang, and K. A. Sakallah. "Generating Data Race Witnesses by an SMT-Based Analysis". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 313–327. DOI: 10.1007/978-3-642-20398-5_23. URL: https://doi.org/10.1007/978-3-642-20398-5%5C_23.

90. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs". *ACM Trans. Comput. Syst.* 15:4, 1997, pp. 391–411. DOI: 10.1145/265924.265927. URL: https://doi.org/10.1145/265924.265927.

91. Y. J. Song, R. van Renesse, F. B. Schneider, and D. Dolev. "The Building Blocks of Consensus". In: *Distributed Computing and Networking, 9th International Conference, ICDCN 2008, Kolkata, India, January 5-8, 2008*. Ed. by S. Rao, M. Chatterjee, P. Jayanti, C. S. R. Murthy, and S. K. Saha. Vol. 4904. Lecture Notes in Computer Science. Springer, 2008, pp. 54–72. DOI: 10.1007/978-3-540-77444-0_5. URL: https://doi.org/10.1007/978-3-540-77444-0%5C_5.

92. S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. "TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs". In: *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*. Ed. by H. Giese and G. Rosu. Vol. 7273. Lecture Notes in Computer Science. Springer, 2012, pp. 219–234. DOI: 10.1007/978-3-642-30793-5_14. URL: https://doi.org/10.1007/978-3-642-30793-5%5C_14.

93. A. Valmari. "Stubborn sets for reduced state space generation". In: *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*. Ed. by G. Rozenberg. Vol. 483. Lecture Notes in Computer Science. Springer, 1989, pp. 491–515. DOI: 10.1007/3-540-53863-1_36. URL: https://doi.org/10.1007/3-540-53863-1%5C_36.

94. M. T. Vechev, E. Yahav, and G. Yorsh. "Abstraction-guided synthesis of synchronization". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by M. V. Hermenegildo and J. Palsberg. ACM, 2010, pp. 327–338. DOI: 10.1145/1706299.1706338. URL: https://doi.org/10.1145/1706299.1706338.

95. M. T. Vechev, E. Yahav, and G. Yorsh. "Inferring Synchronization under Limited Observability". In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by S. Kowalewski and A. Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 139–154. DOI: 10.1007/978-3-642-00768-2_13. URL: https://doi.org/10.1007/978-3-642-00768-2%5C_13.

96. W. Visser, C. S. Pasareanu, and S. Khurshid. "Test input generation with java PathFinder". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*. Ed. by G. S. Avrunin and G. Rothermel. ACM, 2004, pp. 97–107. DOI: 10.1145/1007512.1007526. URL: https://doi.org/10.1145/1007512.1007526.

97. C. Wang, R. Limaye, M. K. Ganai, and A. Gupta. "Trace-Based Symbolic Analysis for Atomicity Violations". In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by J. Esparza and R. Majumdar. Vol. 6015. Lecture Notes in Com-

puter Science. Springer, 2010, pp. 328–342. DOI: 10.1007/978-3-642-12002-2_27. URL: https://doi.org/10.1007/978-3-642-12002-2%5C_27.

98. Z. Wang, C. Zhao, S. Mu, H. Chen, and J. Li. "On the Parallels between Paxos and Raft, and how to Port Optimizations". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by P. Robinson and F. Ellen. ACM, 2019, pp. 445–454. DOI: 10.1145/3293611.3331595. URL: https://doi.org/10.1145/3293611.3331595.

99. D. Weeratunge, X. Zhang, and S. Jagannathan. "Accentuating the positive: atomicity inference and enforcement using correct executions". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Ed. by C. V. Lopes and K. Fisher. ACM, 2011, pp. 19–34. DOI: 10.1145/2048066.2048071. URL: https://doi.org/10.1145/2048066.2048071.

100. J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by D. Grove and S. M. Blackburn. ACM, 2015, pp. 357–368. DOI: 10.1145/2737924.2737958. URL: https://doi.org/10.1145/2737924.2737958.

101. J. M. Wing and C. Gong. "Testing and Verifying Concurrent Objects". *J. Parallel Distrib. Comput.* 17:1-2, 1993, pp. 164–182. DOI: 10.1006/jpdc.1993.1015. URL: https://doi.org/10.1006/jpdc.1993.1015.

102. D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. E. Anderson. "Planning for change in a formal verification of the raft consensus protocol". In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*. Ed. by J. Avigad and A. Chlipala. ACM, 2016, pp. 154–165. DOI: 10.1145/2854065.2854081. URL: https://doi.org/10.1145/2854065.2854081.

103. Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. "Efficient Stateful Dynamic Partial Order Reduction". In: *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*. Ed. by K. Havelund, R. Majumdar, and J. Palsberg. Vol. 5156. Lecture Notes in Computer Science. Springer, 2008, pp. 288–305. DOI: 10.1007/978-3-540-85114-1_20. URL: https://doi.org/10.1007/978-3-540-85114-1%5C_20.

104. X. Yi, J. Wang, and X. Yang. "Stateful Dynamic Partial-Order Reduction". In: *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*. Ed. by Z. Liu and J. He. Vol. 4260. Lecture Notes in Computer Science. Springer, 2006, pp. 149–167. DOI: 10.1007/11901433_9. URL: https://doi.org/10.1007/11901433%5C_9.

105. M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. "HotStuff: BFT Consensus with Linearity and Responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by P. Robinson and F. Ellen. ACM, 2019, pp. 347–356. DOI: 10.1145/3293611.3331591. URL: https://doi.org/10.1145/3293611.3331591.

106. Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. "Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections". In: *Languages and Compilers for Parallel Computing, 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*. Ed. by J. N. Amaral. Vol. 5335. Lecture Notes in Computer Science. Springer, 2008, pp. 141–155. DOI: 10.1007/978-3-540-89740-8_10. URL: https://doi.org/10.1007/978-3-540-89740-8%5C_10.